Quick answers to common problems

Unity 4.x Cookbook

Over 100 recipes to spice up your Unity skills

Matt Smith Chico Queiroz

[PACKT] PUBLISHING

www.it-ebooks.info

Unity 4.x Cookbook

Over 100 recipes to spice up your Unity skills

Matt Smith

Chico Queiroz

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

www.it-ebooks.info

Unity 4.x Cookbook

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2013

Production Reference: 1070613

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-042-3

www.packtpub.com

Cover Image by J. Blaminsky (milak6@wp.pl)

www.it-ebooks.info

Credits

Authors

Matt Smith

Chico Queiroz

Reviewers

Peter Bruun

Jate Wittayabundit

Acquisition Editor

Mary Nadar

Lead Technical Editor

Dayan Hyames

Technical Editors

Dennis John

Dominic Pereira

Project Coordinator

Amey Sawant

Proofreaders

Lindsey Thomas

Joel Johnson

Indexer

Hemangini Bari

Graphics

Abhinash Sahu

Production Coordinator

Prachali Bhiwandkar

Cover Work

Prachali Bhiwandkar

www.it-ebooks.info

About the Authors

Matt Smith is senior lecturer in computing at the Institute of Technology Blanchardstown, Dublin, Ireland (`www.itb.ie`). In 1980 (you do the math) Matt started computer programming (on a ZX80) and has been programming ever since. In 1985, Matt wrote the lyrics, and was a member of the band that played (and sang, sorry about that by the way) the music on the B-side of the audio cassette carrying the computer game Confuzion (`wikipedia.org/wiki/Confuzion`).

Matt holds a bachelor's degree in Business Computing (Huddersfield University, UK), and as that was a bit boring, he went on to get a masters in Artificial Intelligence (Aberdeen University, Scotland), and a PhD in Computational Musicology (Open University, UK). Having run out of money after 10 years as a full-time student, he began his career as a lecturer and academic. He has been lecturing and researching on programming, artificial intelligence, web development, and interactive multimedia for almost 20 years, holding full-time positions at Winchester University and London's Middlesex University, before moving to his present post in Ireland in 2002. In recent years, Matt has replaced Flash-based 2D multimedia with Unity-based 3D game development and interactive virtual environments subjects for his computing and digital media undergraduates.

To keep himself fit, Matt took up the Korean martial art of Taekwon-Do (he developed and runs his club's website at `www.maynoothtkd.com`), and a group of his BSc students are now developing a Unity-based Taekwon-Do interactive "tutor" with Microsoft Kinect cameras. Some of his previous Irish-French student team games can be found and played at `www.saintgermes.com` (thanks for continuing to host these, Guillem!). Matt was one of the two technical experts for a recent multimedia European project for language and cultural student work mobility (`vocalproject.eu`).

www.it-ebooks.info

Matt is currently struggling to learn Korean (for his Taekwon-Do), and Irish (since his daughter Charlotte attends an Irish-speaking school and he doesn't believe her translations of her teacher's report cards ...). In 2012, he started taking classical piano lessons again (after a 20-year gap), with a view to sitting exams starting May, 2013.

Matt's previous authoring includes contributions to *Serious Games and Edutainment Applications*, *Springer* (2011), *Musical Imagery*, *Routledge* (2001). He was also lead editor for *Music Education: An Artificial Intelligence Approach*, *Springer* (1994), and a technical reviewer for *Internet and World Wide Web: How to Program (3rd Edition) by Deitel, Deitel & Goldberg*, *Prentice Hall* (2003).

Thanks to my family for all their support. Thanks also to my students, who continue to challenge and surprise me with their enthusiasm for multimedia and game development.

I would like to dedicate this book to my wife Sinead and children Charlotte and Luke.

---

www.it-ebooks.info

Chico Queiroz is a multimedia designer from Rio de Janeiro, Brazil. Chico initiated his career back in 2000, soon after graduating in Communications/Advertising (PUC-Rio), working with advergames and webgames using Flash and Director at LocZ Multimedia. Here he contributed to the design and development of games for clients, such as Volkswagen and Parmalat, along with some independent titles.

Chico has a Master's Degree in Digital Game Design (University for the Creative Arts, UK). His final project was exhibited at events and festivals, such as London Serious Games Showcase and FILE. Chico has also published articles for academic conferences and websites, such as `gameology.org`, `gamasutra.com`, and `gamecareerguide.com`.

He curated and organized an exhibition, held at SBGames 2009, which explored connections between video games and art. SBGames is the annual symposium of the Special Commission of Games and Digital Entertainment of the Computing Brazilian Society.

Chico currently works as a Digital Designer at the Computer Graphics Technology Group (TecGraf), within the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), where he, among other responsibilities, uses Unity to develop interactive presentations and concept prototypes for interactive visualization software. He also works as a lecturer at PUC-Rio, teaching undergraduate Design students 3D modeling and Technology/CG for Games, in which Unity is used as the engine for the students' projects.

I would like to thank my friends, family, co-workers, and all who have made this book possible and have helped me along the way. Special thanks to Stefano Corazza, Anaïs Gragueb, and Oliver Barraza for their fantastic work at Mixamo; Eduardo Thadeu Corseuil, my manager at TecGraf, for giving me the opportunity of using Unity in our interactive projects. Peter Dam and Peter Hohl from TecGraf, and Paul Bourke from the University of Western Australia, for their help and advice on stereo 3D visualization; Aldo Naletto for sharing his knowledge on sound engineering; my students and colleagues at PUC-Rio Art and Design department.

I would like to dedicate this book to my wife Ana and my daughters Alice and Olivia. Thank you for all your love and support.

---

www.it-ebooks.info

About the Reviewers

Peter Bruun is an independent game developer based in Copenhagen, Denmark. He loves beautiful games and old sci-fi B-movies from the 1950s. For many years Peter has been jumping from project to project as a freelance programmer in the games industry. More recently, he was the lead game programmer on the hit mobile game Subway Surfers, which has been played by millions of people worldwide.

Jate Wittayabundit was an interior architect for several companies in Bangkok, Thailand. Then, he moved to Toronto, Canada and is now a Game Developer and Technical Artist at `Splashworks.com` Inc. and hopes to build his own company in the near future.

He is passionate about gaming and new technology, especially Unity. He is also the author of *Unity 3 Game Development Hotshot*, *Packt Publishing*.

In his spare time, he loves to work on 3D software, such as Zbrush or 3D Studio Max. He also loves painting and drawing. Currently, he's trying to merge his architectural and 3D skills with his game development skills to create the next innovation game.

www.it-ebooks.info

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

www.it-ebooks.info

Table of Contents

Preface 1

Chapter 1: Getting Started with Unity 4.x 7

Introduction 7
Installing Unity 4.x 8
Setting your preferences 9
Understanding and optimizing the User Interface 10
Saving assets created in Unity as Prefabs 13
Discovering Unity's content 15
Importing your own content 16
Importing Unity packages into your project 17
Importing custom packages into your project 19
Exporting custom packages from your project 21
Adding custom packages to Unity's quick list 22
Using the Project browser 24

Chapter 2: Using Cameras 27

Introduction 27
Creating a picture-in-picture effect 27
Switching between multiple cameras 32
Customizing the lens flare effect 35
Making textures from screen content 39
Zooming a telescopic camera 43
Making an inspect camera 46
Creating particle effects using Shuriken 48
Displaying a mini-map 52

Chapter 3: Creating Maps and Materials 59

Introduction 59
Creating a reflective material 60
Creating a self-illuminated material 64

www.it-ebooks.info

Creating specular texture maps 68
Creating transparency texture maps 72
Using cookie textures to simulate a cloudy outdoor 76
Creating a color selection dialog 81
Combining textures in real time through the GUI 84
Highlighting materials at mouse over 87
Animating textures by looping through array of materials (for example, simulated video) 90
Disabling culling for a material 92

Chapter 4: Creating GUIs 95
Introduction 96
Displaying a digital clock 96
Displaying an analogue clock 98
Displaying a compass to show player direction 102
Displaying a radar to indicate relative locations of objects 106
Displaying images for corresponding integers 110
Displaying images for corresponding floats and ranges 112
Displaying a digital countdown timer 116
Displaying a countdown timer graphically (5, 4, 3, 2, 1 – blast off) 117
Displaying a countdown timer graphically as a pie-chart style clock 120
Creating a message that fades away 123
Displaying inventory texts for single object pickups 124
Displaying inventory icons for single object pickups 128
Managing inventories with a general purpose PickUp class 130
Controlling the scrollbar with the mouse wheel 134
Implementing custom mouse cursor icons 136

Chapter 5: Controlling Animations 141
Introduction 141
Configuring a character's Avatar and Idle animation 142
Moving your character with Root Motion and Blend Trees 147
Mixing animations with Layers and Masks 157
Overriding Root Motion via script 164
Adding rigid props to animated characters 171
Making an animated character throw an object 175
Applying ragdoll physics to a character 179
Rotating the character's torso to aim 183

Chapter 6: Playing and Manipulating Sounds 189
Introduction 189
Matching audio pitch to animation speed 189
Adding customizable volume controls 194

www.it-ebooks.info

Simulating a tunnel environment with Reverb Zones 198
Preventing the AudioClip from restarting if already playing 201
Waiting for audio to finish before auto-destructing an object 202
Making a dynamic soundtrack 204

Chapter 7: Working with External Resource Files and Devices 211
Introduction 211
Loading external resource files – by Unity Default Resources 212
Loading external resource files – by manually storing files in Unity's Resources folder 214
Loading external resource files – by downloading files from the Internet 217
Saving and loading player data – using static properties 219
Saving and loading player data – using PlayerPrefs 223
Saving screenshots from the game 225
Control characters in Unity with the Microsoft Kinect using the Zigfu samples 227
Animating your own characters with the Microsoft Kinect controller 230
Homemade mocap by storing movements from the Microsoft Kinect controller 232
Setting up a leaderboard using PHP/MySQL 236

Chapter 8: Working with External Text Files and XML Data 243
Introduction 243
Loading external text files using the TextAsset public variable 244
Loading external text files using C# file streams 245
Saving external text files with C# file streams 248
Loading and parsing external XML files 249
Creating XML text data manually using XMLWriter 252
Creating XML text data automatically through serialization 256
Creating XML text files – saving XML directly to text files with XMLDocument.Save() 261

Chapter 9: Managing Object States and Controlling Their Movements 265
Introduction 266
Controlling cube movement through player controls 266
Controlling object look-at behavior 270
Controlling object-to-object movements (seek, flee, follow at a distance) 272
Controlling object group movement through flocking 279
Firing objects by instantiation with forward velocity 285
Finding a random spawn point 290

www.it-ebooks.info

Finding the nearest spawn point 292
Following waypoints in a sequence 295
Managing object behavior with states 298
Managing complex object behavior with the state pattern 302

Chapter 10: Improving Games with Extra Features and Optimization 309

Introduction 309
Pausing the game 310
Implementing slow motion 312
Implementing 3D stereography with polarized projection 316
Preventing your game from running on unknown servers 321
Identifying performance "bottlenecks" with code profiling 324
Reducing the number of objects by destroying objects at a "death" time 326
Reducing the number of enabled objects by disabling objects whenever possible 328
Improving efficiency with delegates and events (and avoiding SendMessage!) 332
Executing methods regularly but independent of frame rate with coroutines 335
Spreading long computations over several frames with coroutines 337
Caching, rather than component lookups and "reflection" over objects 339

Chapter 11: Taking Advantage of Unity Pro 345

Introduction 345
Dynamically focusing objects with Depth of Field 345
Creating a rearview mirror 348
Playing videos inside a scene 352
Simulating underwater ambience with audio filters 354
Loading and playing external movie files 357

Index 361

www.it-ebooks.info

Preface

Game development is a broad and complex task. An interdisciplinary field covering subjects as diverse as Artificial Intelligence, character animation, digital painting, and sound editing. All those areas of knowledge can materialize as the production of hundreds (or thousands!) of multimedia and data assets. A special software application—the game engine—is required to consolidate all of those assets into a single product.

Game engines are specialized pieces of software, which used to belong to an esoteric domain. They were expensive, inflexible, and extremely complicated to use. They were for big studios or hardcore programmers only. Then along came Unity.

Unity represents true democratization of game development. An engine and multimedia editing environment that is user-friendly and versatile. It has free and indie versions and a Pro version that includes even more features. As we write this preface, Unity offers modules capable of publishing games to Windows, Mac, Linux, iOS, Android, XBox 360, Wii U, and PS3; as well as web-based games using the Unity plugins.

Today, Unity is used by a diverse community of developers all around the world. Some are students and hobbyists, but many are commercial organizations ranging from garage developers to international studios, using Unity to make a huge number of games—some you might have already played in one platform or another.

This book provides over 100 Unity game development recipes. Some recipes demonstrate Unity application techniques for multimedia features, including working with animations and using preinstalled package systems. Other recipes develop game components with C# scripts, ranging from working with data structures and data file manipulation, to artificial intelligence algorithms for computer controlled characters.

If you want to develop quality games in an organized and straightforward way, and want to learn how to create useful game components and solve common problems, then both Unity and this book are for you.

www.it-ebooks.info

What this book covers

Chapter 1, *Getting Started with Unity 4.x*, is written for those who have just started, or are about to start, using Unity 4.x. It covers software installation, interface concepts, user preferences, and some workflow tips.

Chapter 2, *Using Cameras*, will explain recipes covering techniques for controlling and enhancing your game's camera. This chapter will present interesting solutions to work with both single and multiple cameras.

Chapter 3, *Creating Maps and Materials*, contains recipes that will give you—whether you are a game artist or not—a better understanding on how to use maps and materials in Unity 4.x. It should be a great resource for exercising your image editing skills.

Chapter 4, *Creating GUIs*, is filled with GUI (Graphical User Interface) recipes to help you increase the entertainment and enjoyment of your games through the quality of the interactive visual elements. You'll learn a wide range of GUI techniques, including working with scroll wheels for input, and displaying directional compasses, radars, and graphical inventory icons.

Chapter 5, *Controlling Animations*, demonstrates focusing on character animation, how to take advantage of Unity's new animation system—Mecanim. It covers everything from basic character setup to procedural animation and ragdoll physics.

Chapter 6, *Playing and Manipulating Sounds*, is dedicated to making sound effects and soundtrack music in your game more interesting. It also touches on playback and volume control techniques.

Chapter 7, *Working with External Resource Files and Devices*, throws light on how external data can enhance your game in ways, such as adding renewable content and communicating with websites. External devices, such as the Microsoft Kinect, can totally change the game's interactions. Learn about communicating with external resources and devices in this chapter.

Chapter 8, *Working with External Text Files and XML Data*, provides recipes for different methods to work with text files in general, and with XML text data specifically. This chapter is included because XML and other text-based data is common and very useful, both being computer and human readable.

Chapter 9, *Managing Object States and Controlling Their Movements*, relates to the many games that involve moving computer-controlled objects and characters. For many games animation components can be sufficient. However, other games use artificial intelligence for directional logic. This chapter presents a range of such directional recipes, which can lead to games with a richer and more exciting user experience.

www.it-ebooks.info

Chapter 10, *Improving Games with Extra Features and Optimization*, provides several recipes providing some ideas for adding some extra features to your game (pausing, slow motion, 3D stereography, and securing online games). The rest of the recipes in this chapter provide examples of how to investigate and improve the efficiency and performance of your game's code.

Chapter 11, *Taking Advantage of Unity Pro*, is a concise chapter with interesting uses for some Unity Pro capabilities. It includes recipes for sound, render texture, video texture, and image effects.

What you need for this book

You will need a copy of Unity 4.x, which can be downloaded for free from `http://www.unity3d.com`. If you wish to create your own image files for the recipes in *Chapter 3*, *Creating Maps and Materials*, you will also need an image editor such as Adobe Photoshop (which can be found at `http://www.photoshop.com`) or GIMP, which is free and can be found at `http://www.gimp.org`.

Who this book is for

This book is for anyone who wants to explore a wide range of Unity scripting and multimedia features and find ready-to-use solutions to many game features. Programmers can explore multimedia features, and multimedia developers can try their hand at scripting.

From beginners to advanced users, from artists to coders, this book is for you and everyone in your team!

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Unity's menu actually reads the `Standard Packages` folder content when starting up, instead of getting that information from somewhere else."

A block of code is set as follows:

```
private void ChangeMaterial()  {
  materialIndex++;
  materialIndex = (materialIndex % materialArray.Length);
  Material nextMaterial = materialArray[ materialIndex ];
  renderer.sharedMaterial = nextMaterial;
}
```


www.it-ebooks.info

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "Let's create a new material. Access the **Project** view, click on the **Create** drop-down menu and choose **Material**. Rename it to `Grid`."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

www.it-ebooks.info

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

www.it-ebooks.info

www.it-ebooks.info

1
Getting Started with Unity 4.x

In this chapter, we will cover:

- Installing Unity 4.x
- Setting your preferences
- Understanding and optimizing the User Interface
- Saving assets created in Unity as Prefabs
- Discovering Unity's content
- Importing your own content
- Importing Unity packages into your project
- Importing custom packages into your project
- Exporting custom packages from your project
- Adding custom packages to Unity's quick list
- Using the Project browser

Introduction

This chapter is tailored for those who are about to start using Unity or have just arrived to it. In this chapter, you will find some introductory steps into making this engine more comfortable and familiar.

www.it-ebooks.info

Installing Unity 4.x

Unity is a very powerful and versatile game engine. It is available in both Indie (free) and Pro (paid) versions. In case you haven't installed Unity yet, this recipe will show you how to do it.

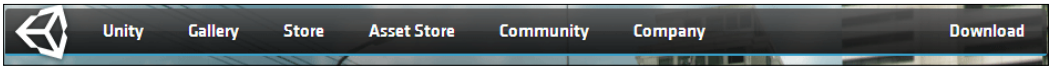
Getting ready...

You will need Internet access to follow this recipe.

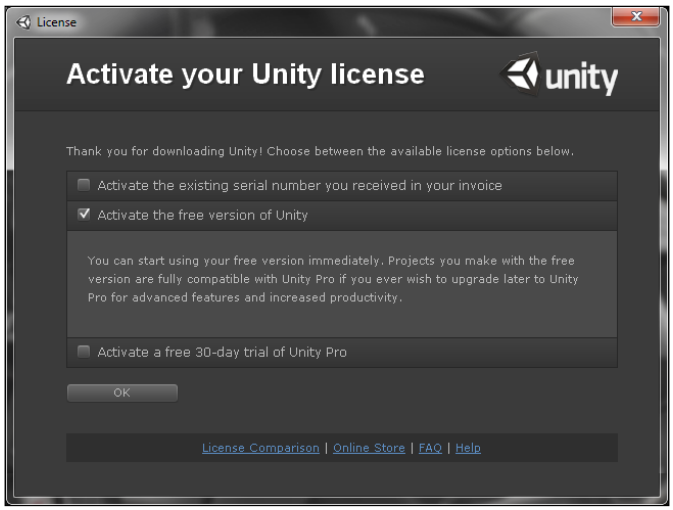
How to do it...

To Install Unity, please follow these steps:

1. Access the Unity website at `www.unity3d.com`.
2. Locate and click the **Download b**utton, placed in the top-right corner.



3. Now, on the **Download** page, click the button to get the latest version of Unity. Wait for the download to complete.
4. Run the Installer. This is a very straightforward process that will install everything you need in a couple of minutes.
5. Once the software is installed, run Unity. That should take you to the activation dialog where you can choose between activating Unity Pro (provided you have a valid serial number), Unity Free, or a 30-day trial of Unity Pro:



www.it-ebooks.info

6. Select your choice and click **OK**. You should be prompted to log in or create an account.
7. Register (if necessary) and log in to activate your copy of Unity and start using it right away.

There's more...

You can expand Unity's capabilities and reach new audiences by adding more platforms to your editor.

Acquiring new licenses

iOS and Android exporters are now included free, and others can be bought from the Unity Store at `https://store.unity3d.com`.

Setting your preferences

Setting the editor to your preferences might sound superfluous to some. However, it could accelerate your development process and make Unity even more comfortable to use. In this recipe, we will learn how to adjust some of those settings to your taste.

How to do it...

To adjust Unity's preferences, follow these steps:

1. Inside the Unity editor, navigate to **Edit** | **Preferences...** (or, if you are using Mac OS, **Unity** | **Preferences...**).
2. As the **Preferences** window shows up, notice that it is divided into these sections: **General**, **External Tools**, **Colors, Keys**, and **Cache Server**.
3. Select the **General** tab. If you're working with multiple projects, you might want to leave the **Always Show Project Wizard** option checked.
4. Also, if you use OS X and are used to its native color picker, leave the **OSX Color Picker** option checked.
5. Now select the **External Tools** tab. In case you want to use a different script editor than Unity's built-in MonoDevelop, you can use the drop-down menu in **External Script Editor** to browse to your favorite application.
6. If **Image Application** is set as **Open by File Extension**, you might end up working with several image editors simultaneously. To avoid that, use the drop-down menu to browse to your favorite software.
7. Also, if you happen to develop to Android, make sure to browse to the SDK in **Android SDK Location**.

www.it-ebooks.info

8. Let's move on to **Colors** tab. The default settings are fine, but feel free to change colors that make you most comfortable.
9. Now select the **Keys** tab. You might select any action to change its shortcut. Again, the default settings are perfectly fine. Use this opportunity to learn more about them.

There's more...

As you probably noticed, Unity's **Preferences** window has more options than was covered here. If you want a full explanation for each setting, please check the online documentation at `http://docs.unity3d.com/Documentation/Manual/Preferences.html`.

Changing the editor's player quality settings

Depending on your target platform, you might want to adjust the level of graphical quality of your game. This can be done through **Quality Settings**, which controls, for instance, the resolution of real-time shadows, or how much anti aliasing will be applied. Those options (and much more) are organized in levels that range from **Fastest** to **Fantastic**. If you want to experience a particular quality setting when running your game from the Editor, navigate to **Edit** | **Project Settings** | **Quality** and select it from the table in the **Inspector** view.

See also

- The *Understanding and optimizing the User Interface* recipe.

Understanding and optimizing the User Interface

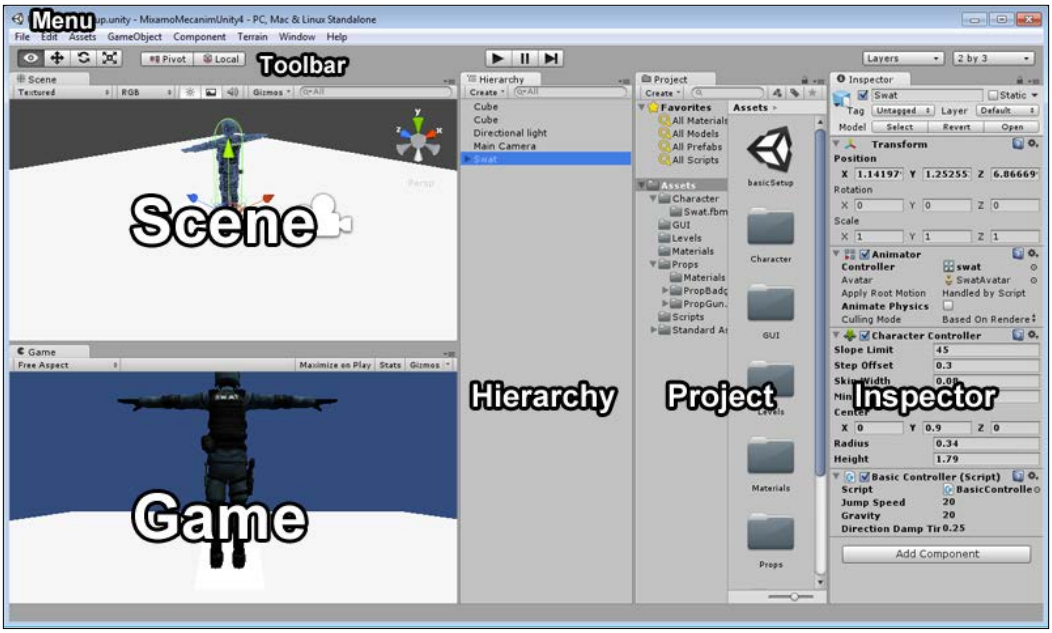
Game engines, especially 3D-capable ones, can be a bit intimidating the first time you open them. Although Unity is particularly intuitive, user-friendly, and well documented, we have provided this recipe to show you how to operate inside its User Interface (UI).

How to do it...

Let's take a look at Unity's user interface:

1. Run Unity. Unless you have previously changed it, its layout should initiate in Wide mode. Access **Window** | **Layouts** and choose another option, such as **4 Split** or **2 by 3**, and notice how the interface is organized into **Views**:

www.it-ebooks.info

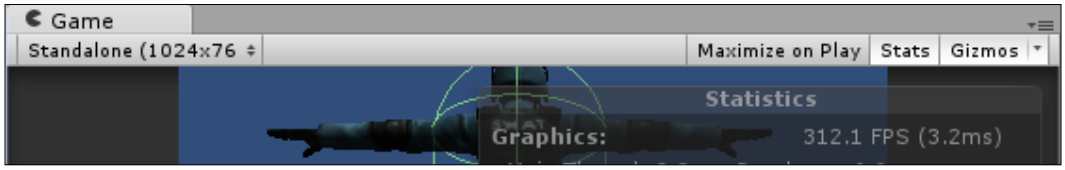


Let's take a look at those views:

- **Scene:** This view is used to position, rotate, scale, and select game objects, and also navigate your level.
- **Game:** This is the place to play and test your game. It will reproduce the player's experience as accurately as possible.
- **Hierarchy**: Game objects (as diverse as characters, cameras, level geometry, lights, and even GUI textures) placed in our scene will be listed here.
- **Project**: This is where you create, organize, and access your game assets. From 3D models and 2D textures to C# scripts and Prefabs, every re-usable element will be listed here.
- **Inspector**: From the Inspector, you can configure any game objects (selected from the Hierarchy view) or assets (selected from the Project view). That includes changing its **Transform** settings, configuring existing components and attaching new ones. Also, you can adjust other preferences for your game, once you have accessed them from the menu, in the Inspector view.
- **Toolbar**: Includes transform tools (used for manipulating game objects and navigating the scene), control tools (used for playing / pausing and stopping the level), and drop-down tools (used for managing layers and layouts).
- **Menu**: Gives access to a diverse list of commands covering asset import/export, preferences setting, game object creation, components, terrain, layout, and documentation.

www.it-ebooks.info

2. If you want to customize the layout any further, drag and drop the views to relocate and/or dock them.
3. If you like your custom layout, save it through the **Window** | **Layouts** | **Save Layout...** menu.
4. When testing your game, it might be a good idea to check the **Maximize on Play** button, in the **Game** view. Also, if you work with more than one display monitor, you could drag the **Game** view into the second display, leaving a display exclusively for the Editor.
5. You can also adjust the **Game** view resolution. It's a good idea to test your game running on its standard standalone resolution and every supported aspect ratio.
6. In case you want to check the graphics performance of your game during testing, you should turn on the **Stats** button (you can also turn it off during testing, if so you wish).
7. Finally, activate **Gizmos** if you want them to be drawn at runtime, making it easier to spot rays, colliders, lights, cameras, and so on in your scene, as shown here:



8. There is another view you should pay attention to: the **Console** view. Access it by navigating to **Window** | **Console**. This is a very important view when it comes to debugging your game, as it displays errors, warnings, and other debug output during testing.
9. Another interesting view (for those with Unity Pro) is the **Profiler** (**Window** | **Profiler**), where you can check out detailed statistics of your game performance in real time.

There's more...

To get an extensive explanation on each UI feature, please check out Unity's documentation at `http://docs.unity3d.com/Documentation/Manual/LearningtheInterface.html`.

See also

- The *Setting your preferences* recipe.
- The *Searching assets with the Project browser* recipe.

www.it-ebooks.info

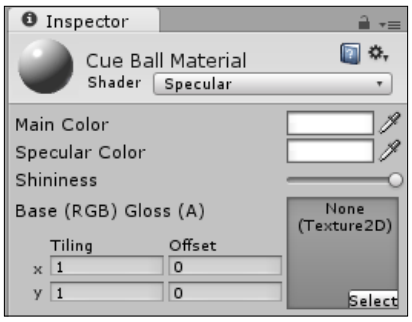
Saving assets created in Unity as Prefabs

You can easily create primitive geometry with Unity. In this recipe, we will create a game object from Unity's resources and keep it in our project as a **Prefab**.

How to do it...

To create a Prefab, follow these steps:

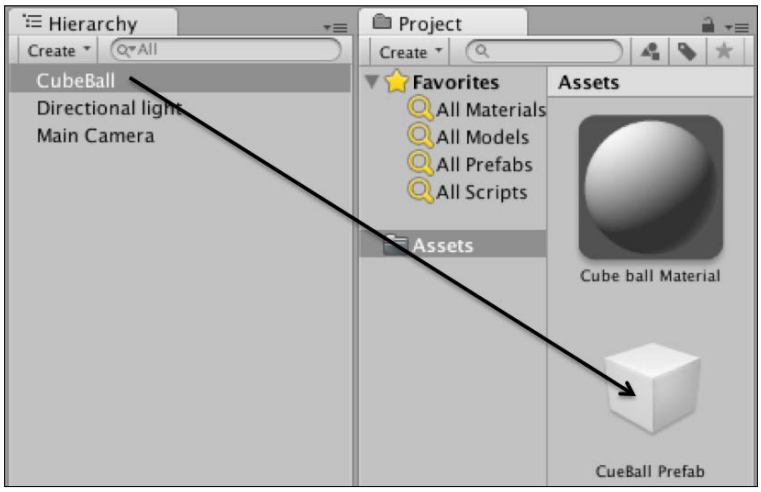
1. Inside the Unity editor, navigate to **GameObject** | **Create Other** | **Sphere**.
2. In the **Hierarchy** view, right-click **Sphere** and choose the appropriate option from the context menu to rename it to **Cue Ball**.
3. Now, in the **Project** view, click on the **Create** button and choose the **Material** option. Then, rename the new material to **Cue Ball Material**.
4. In the **Project** view, select **Cue Ball Material**. Then, in the **Inspector** view, change its **Shader** value to **Specular**.
5. Also, set **Specular Color** to white and set its **Shininess** to the maximum, as shown in the following screenshot:



6. From the **Project** view, drag **Cue Ball Material** into the **Cue Ball** game object, in the **Hierarchy** view.
7. Select **Cue Ball**. Then, access **Component** | **Physics** | **Rigidbody**. That should attach a **Rigidbody** component to that game object.
8. Now that your game object is complete, click on the **Create** button in the **Project** view and choose the **Prefab** option. Then, rename it to **Cue Ball Prefab**.

www.it-ebooks.info

9. Drag the **Cue Ball** game object from the **Hierarchy** view into the Prefab in the **Project** view. Your game object is ready to be re-used in this project.



How it works...

In Unity, game objects can be stored as Prefabs. This is very useful in case you want to re-use a game object in several levels or instantiate it through scripting. Adobe Flash users can think of it as the Unity equivalent of MovieClips.

There's more...

There are many other ways to use Unity's built-in resources. Here are some ideas:

Adding external files

In this recipe, we haven't used any external asset. However, there's no reason you could not have imported a texture and used it as the `Cue Ball Material` base map, for instance.

Taking your Prefabs to another project

Also, if you plan of re-using your Prefab in other projects, you can do it by exporting it as a custom package.

Creating other kinds of game objects

As you have probably noticed, spheres are not the only entities you can create directly with Unity. Other primitives are also available, as well as many other types of entities: lights, camera, GUI textures, and so on. Navigate to **GameObject** | **Create Other** and experiment with the options available.

www.it-ebooks.info

See also

- The *Exporting Custom packages from your project* recipe.

Discovering Unity's content

As you enter Unity for the first time, you might think you'll need to build and code everything from scratch. However, Unity comes with several collections of content called **Packages**, designed to save you time when implementing commonly required features.

How to do it...

Let's find out what's inside Unity's standard packages:

1. Inside the Unity editor, access the **Assets** menu.
2. Expand the **Import Package** submenu.
3. You will see a list of available packages from Unity. These are filled with ready-to-use content.

How it works...

Unity makes implementing commonly requested features easy by making them available as packages ready to be imported and used in your project. These packages include First-Person and Third-Person Character Controllers, Image Effects (Pro Only), Terrain textures and assets, Skyboxes, Water, Tree Creator, and more.

There's more...

There are other ways to learn from ready-made material. Here are some:

Studying the sample project

Unity also comes with a sample project ready to be dissected by you. It automatically opens the first time you start the software.

Downloading more resources

You can find and download even more resources, such as packages, projects, tutorials and assets, from Unity's resources page at `http://unity3d.com/support/resources/`.

See also

- The *Importing Custom packages into your project* recipe.

www.it-ebooks.info

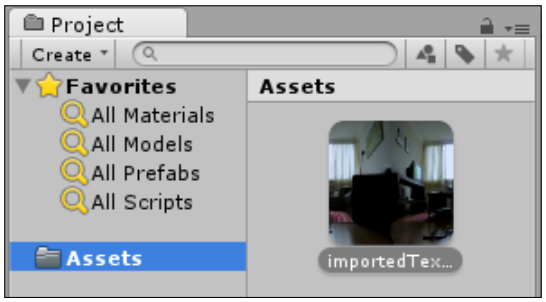
Importing your own content

After you have created a 3D model, audio clip, movie clip, or texture, you can import it into your project. In this recipe, we will learn how it can be done.

How to do it...

Follow these steps to import an asset:

1. Inside the Unity editor, access the **Assets** menu.
2. Select the **Import New Asset...** option.
3. Browse to your file and click **Import**.
4. Your file should be now listed in the **Project** browser, as shown here:



How it works...

Unity makes a copy of your file, converts it to an appropriate format (if necessary) and saves it into the **Project** browser's **Assets** folder.

There's more...

Here's a couple of helpful pieces of information on the subject:

Organizing it with the Project view

Unity updates its **Project** view whenever a new file is added to the **Assets** folder. You could then save or export your work directly into that folder. You could also paste or move multiple files into there. However, you should not reorganize or rename your imported files via your OS file management system (Window's Explorer or Mac OS Finder), as this could damage important information kept by Unity about those files.

www.it-ebooks.info

Exporting your assets to Unity 4.x

If you are not sure about how to prepare and export your work to Unity, or which file format you should use, please check out Unity's documentation at `http://docs.unity3d.com/Documentation/Manual/AssetImportandCreation.html` for a very comprehensive guide on the subject. Some other useful pages regarding the subject are:

- Importing objects from 3D Studio Max: `http://docs.unity3d.com/Documentation/Manual/HOWTO-ImportObjectMax.html`
- Importing objects from Maya: `http://docs.unity3d.com/Documentation/Manual/HOWTO-ImportObjectMaya.html`
- Export FBX – How-to: `http://docs.unity3d.com/Documentation/Manual/HOWTO-exportFBX.html`

See also

- The *Importing Custom packages into your project* recipe.

Importing Unity packages into your project

The packages provided by Unity can save you a lot of development time. They usually contain resources (such as texture maps, materials, and so on) and fully implemented features ready to go into your project. When creating a new project, Unity offers to install those packages into the **Assets** folder. However, if you've missed it at first, you can still import them into your project at any time.

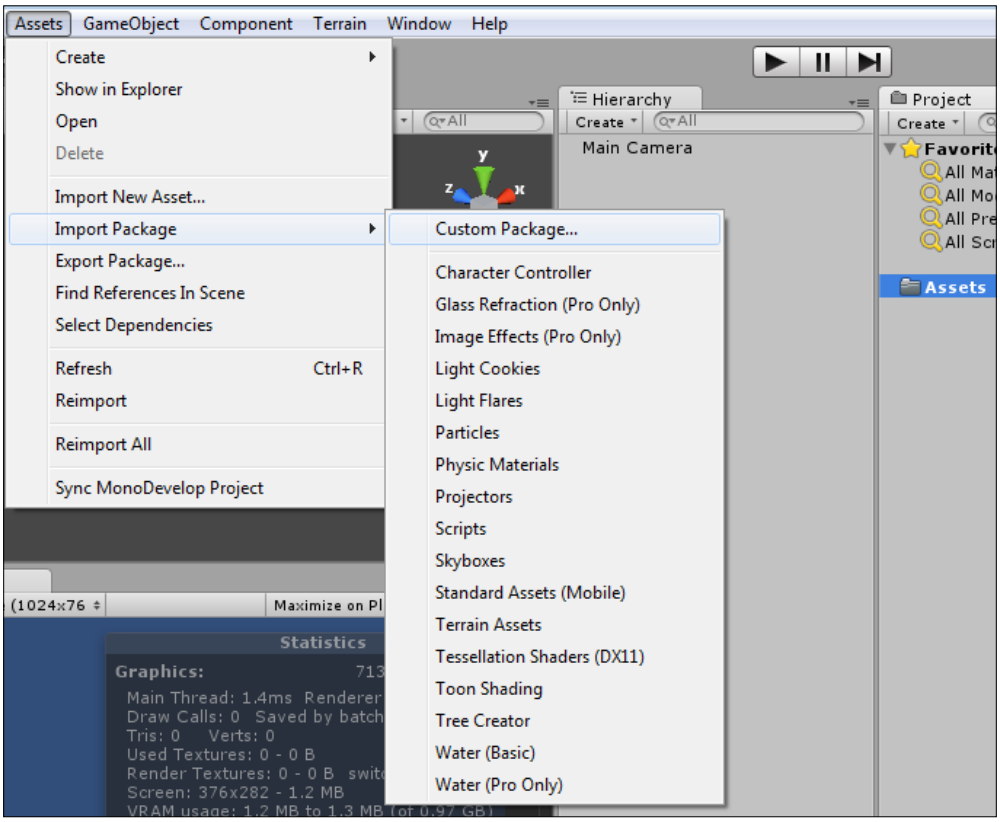
How to do it...

To import a Unity package, follow these steps:

1. Inside the Unity editor, access the **Assets** menu.
2. Enter the **Import Package** sub-menu and choose a package from the list.
3. Make sure every needed component is selected and click **Import**.

www.it-ebooks.info

4. Package contents should be ready and listed in the **Project** view.



How it works...

Unity installation files include a number of packages that can be imported into your project as ready-to-use resources. Inside those packages are all the assets needed to implement a specific feature or functionality. Once imported, new features can be accessed through the **Project** view (and dragged and dropped into your level) or through newly added menu items (the **Tree Creator** package, for instance, adds the **Tree** option into the **Create Other** sub-menu of the **GameObject** menu).

www.it-ebooks.info

There's more...

Importing Unity packages can also be done through the Project Wizard. When starting a new project, check the boxes of the packages you want to import.

See also

- The *Importing custom packages into your project* recipe.
- The *Exporting custom packages from your project* recipe.
- The *Adding custom packages to Unity's quick list* recipe.

Importing custom packages into your project

Custom Unity packages are available from a variety of sources, and they can be very helpful when developing a project.

Getting ready

As its title implies, this recipe requires a custom package to be imported. If you need one for testing purposes, please use the one inside the folder named `0423_01_09-11`.

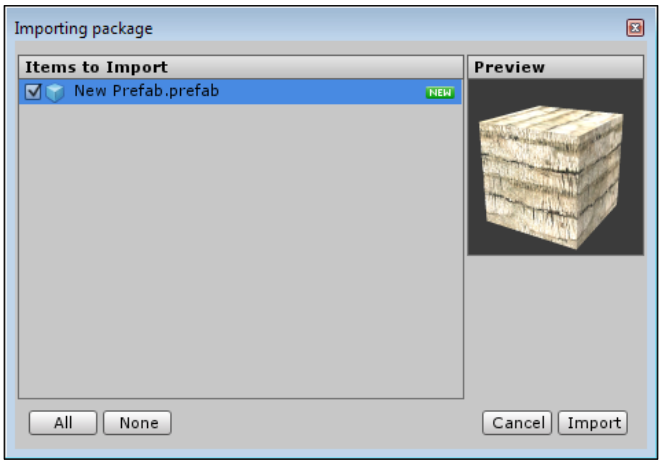
How to do it...

To import a custom package, follow these steps:

1. Inside the Unity editor, access the **Assets** menu.
2. Enter the **Import Package** sub-menu and choose the **Custom Package...** option.
3. Browse to the package file you have saved on your disk and click **Open**.
4. Preview package contents in the top-right **Preview** window, if you like.
5. Make sure every needed component is selected and click **Import**.

www.it-ebooks.info

6. Package contents should be ready and listed in the **Project** view.



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

How it works...

Custom packages are commonly used to distribute a number of assets inside a single compressed file. As they are made by third parties, the content inside those packages may vary, as they can include scripts, 3D models, texture maps, materials and any other file handled by Unity. Once imported, the package content is uncompressed into your project's **Assets** folder, and can be accessed through the **Project** window.

There's more...

Third-party made content can also be found, downloaded, and bought from the Unity Asset Store. For more information, access `http://unity3d.com/unity/asset-store/`.

See also

- The *Importing Unity packages into your project* recipe.
- The *Exporting custom packages from your project* recipe.
- The *Adding custom packages to Unity's quick list* recipe.

www.it-ebooks.info

Exporting custom packages from your project

Creating packages can be a very useful and practical way of storing your game objects and assets for future use and reference. If you want to save a feature, a group of assets, or even a prefab from the project you are currently working on, it's a good idea to export them as a package, so you can easily import them into your future projects.

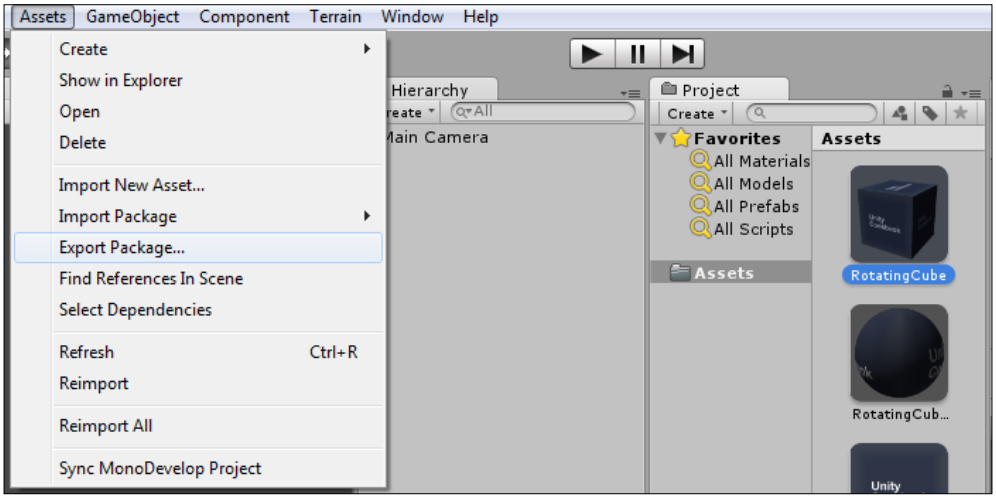
Getting ready

In order to export a package, you will need a project containing some assets. If you need one for testing purposes, please use the one inside the downloadable content for this book, which can be found inside the `0423_01_10` folder.

How to do it...

To export content as a custom package, follow these steps:

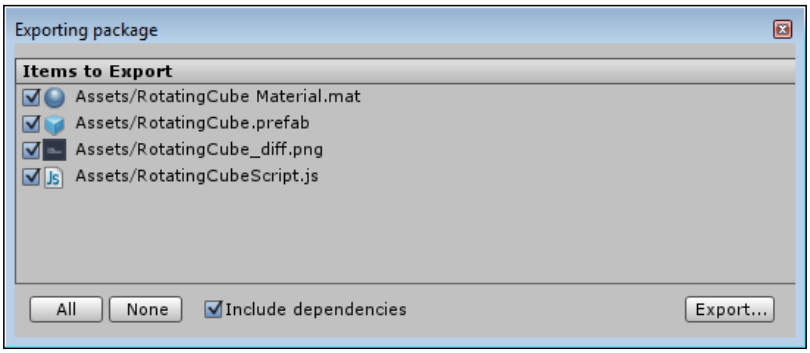
1. Select the **RotatingCube** prefab in the **Project** view.
2. Go to the **Assets** menu and choose the **Select Decencies** option. This will highlight, inside the **Project** tab, all assets that are linked to the **RotatingCube** prefab.
3. Once again, select only the **RotatingCube** prefab.
4. Go to the **Assets** menu and choose **Export Package...**. A new window will now pop up:



5. Inside the **Exporting Package** window, make sure the **Include dependencies** checkbox is selected. It is important that checkboxes for all listed objects are also selected.

www.it-ebooks.info

6. Click **Export** and save your package into your disk. You can give it any name you want (although a name similar to **RotatingCube** will make things easier later, when you want to use it).
7. Your custom package is ready to be imported.



How it works...

By exporting a package, you have stored your selected objects and dependencies into a single compressed file. Importing them to your project will uncompress them into its **Assets** folder.

See also

- The *Importing Unity packages into your project* recipe.
- The *Importing custom packages into your project* recipe.
- The *Adding custom packages to Unity's quick list* recipe.

Adding custom packages to Unity's quick list

If you have one or more packages you'd like to include frequently in your projects, it might be a good idea to add them to Unity's package quick list.

Getting ready

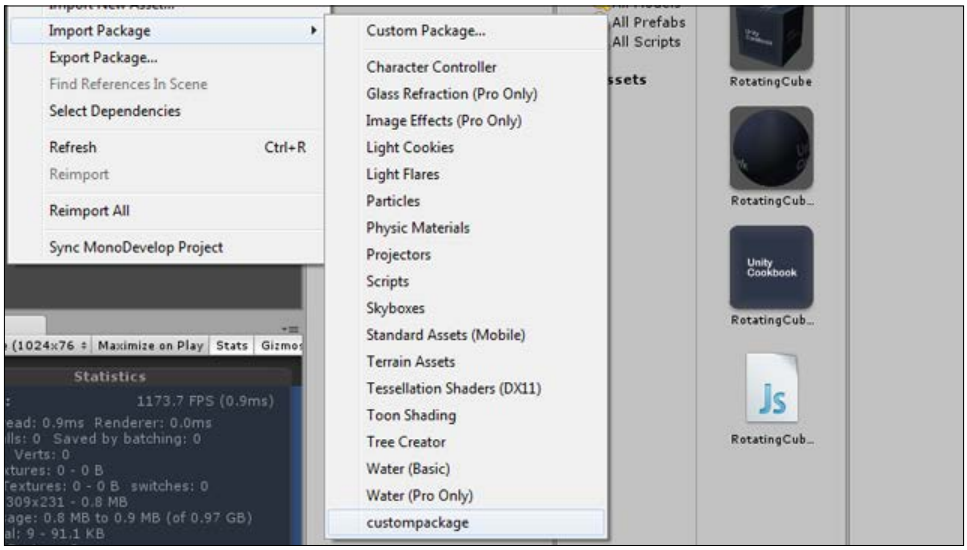
In order to complete this recipe, you will need a custom package (any package will do). If you need one for testing purposes, please use the one inside the `0423_01_09-11` folder.

www.it-ebooks.info

How to do it...

To add a custom package to the quick list, follow these steps:

1. Using your file manager (Windows Explorer on Windows, Finder on Mac OS), browse to the package file and copy it by pressing *Ctrl* + *C* or *Command* + *C*.
2. Go to Unity's `Editor` folder. On Windows, that would typically be `C:/Program Files (x86)/Unity/Editor` or `C:\Program Files\Unity\Editor`. On Mac OS, it should be `Applications/Unity`.
3. Access the `Standard Packages` folder.
4. Paste the previously copied package into this folder.
5. Restart Unity. It should now be listed on the **Assets** menu, under the **Import Package** sub-menu, as shown in the following screenshot:



How it works...

Unity's menu actually reads the `Standard Packages` folder content when starting up, instead of getting that information from somewhere else. This is very practical, as it always reflects the actual content of that folder and also allows the user to quickly retrieve his favorite packages.

There's more...

Custom packages stored in the `Standard Packages` folder will also appear in the **Create New Project Wizard** window, making it simple to add them to new projects.

www.it-ebooks.info

See also

- The *Importing Unity packages into your project* recipe.
- The *Importing custom packages into your project* recipe.
- The *Exporting custom packages from your project* recipe.

Using the Project browser

It doesn't matter how organized you keep your project folders, there will be times when you will need to search for one or more specific assets. To make things easier, Unity 4 includes the **Project** browser. In this recipe, we will learn how to save time by using it.

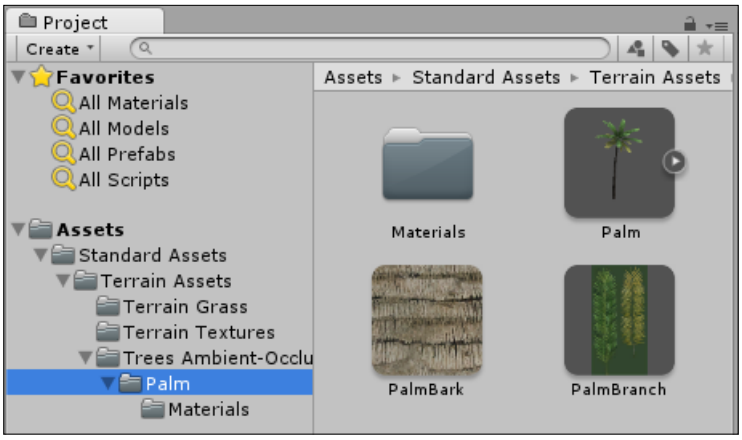
Getting ready

All we need to follow this recipe is a collection of assets. We will use Unity's **Terrain Assets** to populate our **Project** view.

How to do it...

Let's take a look at the **Project** browser:

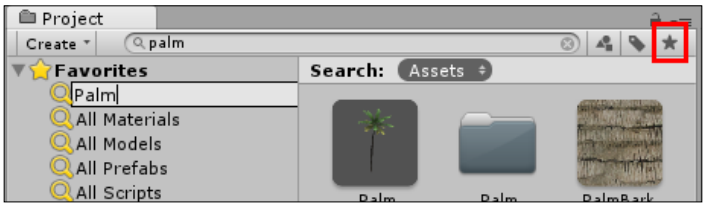
1. Import the **Terrain Assets** package (**Assets** | **Import Package** | **Terrain Assets**).
2. Browse through the **Assets** subfolders to see the files that have been imported and how they are organized, as shown in the following screenshot:



3. Now let's search for all the assets containing the word "Palm" in their names. On the search field located above, type in `Palm`. Observe how every file and folder is listed on the search results window.

www.it-ebooks.info

4. Click on the **Save Search** button (the one with the star icon) and name your search as `Palm`.



5. We could filter our search results by tag or file type. Click the **Search by Type** button (the one with a circle, triangle, and square) and select only the **Models** and **Textures** categories. You can select multiple options by holding *Ctrl* (Windows) or *Command* (Mac).
6. Click on the **Save Search** button and save it as a new filter (name it `Palm Models and Textures`).
7. Browse through the other filters saved in **Favorites**—they are shortcuts to browse through specific file types (**All Materials**, **All Models**, and so on).

How it works...

Unity's new browsing system scans through the assets and lets you save and organize your search results.

See also

- The *Understanding and optimizing the User Interface* recipe.

www.it-ebooks.info

www.it-ebooks.info

2
Using Cameras

In this chapter, we will cover:

- Creating a picture-in-picture effect
- Switching between multiple cameras
- Customizing the lens flare effect
- Making textures from screen content
- Zooming a telescopic camera
- Making an inspect camera
- Creating particle effects using Shuriken
- Displaying a mini-map

Introduction

As developers, we should never forget to pay attention to the cameras. After all, they are the windows from which our players see our game. In this chapter, we will take a look at ways of making them more interesting within the player experience.

Creating a picture-in-picture effect

Having more than one viewport displayed can be useful in many situations. For example, you might want to show simultaneous events going on in different locations, or maybe you want to have a separate window for hot-seat multiplayer games. Although you could do it manually by adjusting the **Normalized Viewport Rect** parameters on your camera, this recipe includes a series of extra preferences to make it more independent from the user's display configuration.

www.it-ebooks.info

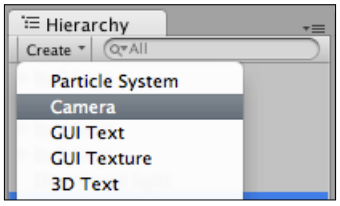
Getting ready

For this recipe, we have prepared a package named `basicLevel` containing a scene. The package is in the `0423_02_01_02` folder.

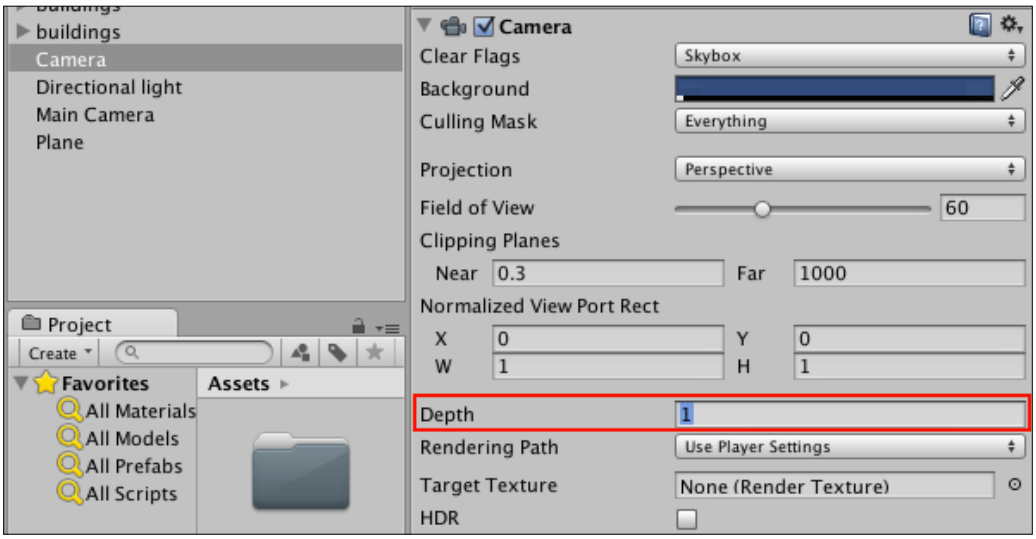
How to do it...

To create a picture-in-picture display, just follow these steps:

1. Import the `basicLevel` package into your Unity project.
2. In the **Project** view, open **basicScene**, inside the folder `02_01_02`. This is a basic scene featuring a directional light, a camera, and some geometry.
3. Add the **Camera** option to the scene through the **Create** dropdown menu on top of the **Hierarchy** view, as shown in the following screenshot:



4. Select the camera you have created and, in the **Inspector** view, set its **Depth** to **1**:



5. In the **Project** view, create a new C# script and rename it `PictureInPicture`.

www.it-ebooks.info

6. Open your script and replace everything with the following code:

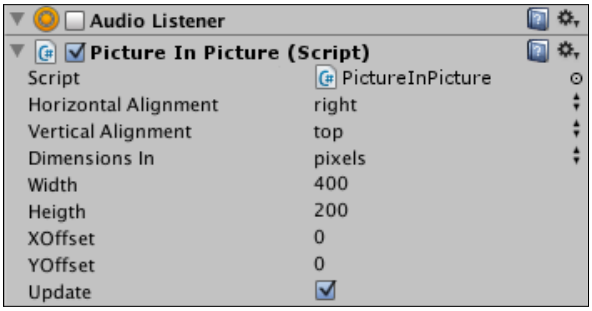
```
using UnityEngine;

public class PictureInPicture: MonoBehaviour {
    public enum HorizontalAlignment{left, center, right};
    public enum VerticalAlignment{top, middle, bottom};
    public HorizontalAlignment horizontalAlignment =
HorizontalAlignment.left;
    public VerticalAlignment verticalAlignment =
VerticalAlignment.top;
    public enum ScreenDimensions{pixels, screen_percentage};
    public ScreenDimensions dimensionsIn = ScreenDimensions.
pixels;
    public int width = 50;
    public int height= 50;
    public float xOffset = 0f;
    public float yOffset = 0f;
    public bool  update = true;
    private int hsize, vsize, hloc, vloc;

    void Start (){
          AdjustCamera ();
    }
    void Update (){
          if(update)
                AdjustCamera ();
    }

    void AdjustCamera(){
          if(dimensionsIn == ScreenDimensions.screen_percentage){
                hsize = Mathf.RoundToInt(width * 0.01f * Screen.
width);
                vsize = Mathf.RoundToInt(height * 0.01f * Screen.
height);
          } else {
                hsize = width;
                vsize = height;
          }

          if(horizontalAlignment == HorizontalAlignment.left){
                  hloc = Mathf.RoundToInt(xOffset * 0.01f *
Screen.width);
            } else if(horizontalAlignment == HorizontalAlignment.
right){
```


www.it-ebooks.info

```
                    hloc = Mathf.RoundToInt((Screen.width - hsize)
- (xOffset * 0.01f * Screen.width));
                } else {
                    hloc = Mathf.RoundToInt(((Screen.width * 0.5f)
- (hsize * 0.5f)) - (xOffset * 0.01f * Screen.height));
                }

            if(verticalAlignment == VerticalAlignment.top){
                    vloc = Mathf.RoundToInt((Screen.height -
vsize) - (yOffset * 0.01f * Screen.height));
                } else if(verticalAlignment == VerticalAlignment.
bottom){
                    vloc = Mathf.RoundToInt(yOffset * 0.01f *
Screen.height);
                } else {
                    vloc = Mathf.RoundToInt(((Screen.height *
0.5f) - (vsize * 0.5f)) - (yOffset * 0.01f * Screen.height));
                }
            camera.pixelRect = new Rect(hloc,vloc,hsize,vsize);

    }
}
```

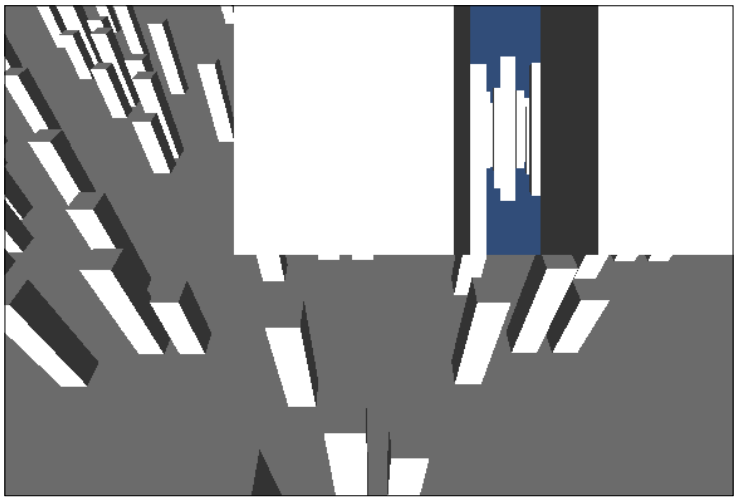
In case you haven't noticed, we are not achieving percentage by dividing numbers by 100, but rather multiplying them by 0.01. The reason behind that is performance: computer processors are faster multiplying than dividing.

7. Save your script and attach it to the new camera that you created previously.
8. Uncheck the new camera's **Audio Listener** component and change some of the **PictureInPicture** parameters: change **Horizontal Alignment** to **Right**, **Vertical Alignment** to **Top**, and **Dimensions In** to **pixels**. Leave **XOffset** and **YOffset** as **0**, change **Width** to **400** and **Height** to **200**, as shown below:



www.it-ebooks.info

9. Play your scene. The new camera's viewport should be visible on the top right of the screen:



How it works...

Our script changes the camera's **Normalized Viewport Rect** parameters, thus resizing and positioning the viewport according to the user preferences.

There's more...

The following are some aspects of your picture-in-picture you could change.

Making the picture-in-picture proportional to the screen's size

If you change the **Dimensions In** option to `screen_percentage`, the viewport size will be based on the actual screen's dimensions instead of pixels.

www.it-ebooks.info

Changing the position of the picture-in-picture

Vertical Alignment and **Horizontal Alignment** can be used to change the viewport's origin. Use them to place it where you wish.

Preventing the picture-in-picture from updating on every frame

Leave the **Update** option unchecked if you don't plan to change the viewport position in running mode. Also, it's a good idea to leave it checked when testing and then uncheck it once the position has been decided and set up.

See also

- The *Displaying a mini-map* recipe.

Switching between multiple cameras

Choosing from a variety of cameras is a common feature in many genres: race sims, sports sims, tycoon/strategy, and many others. In this recipe, we will learn how to give players the ability of choosing an option from many cameras using their keyboard.

Getting ready

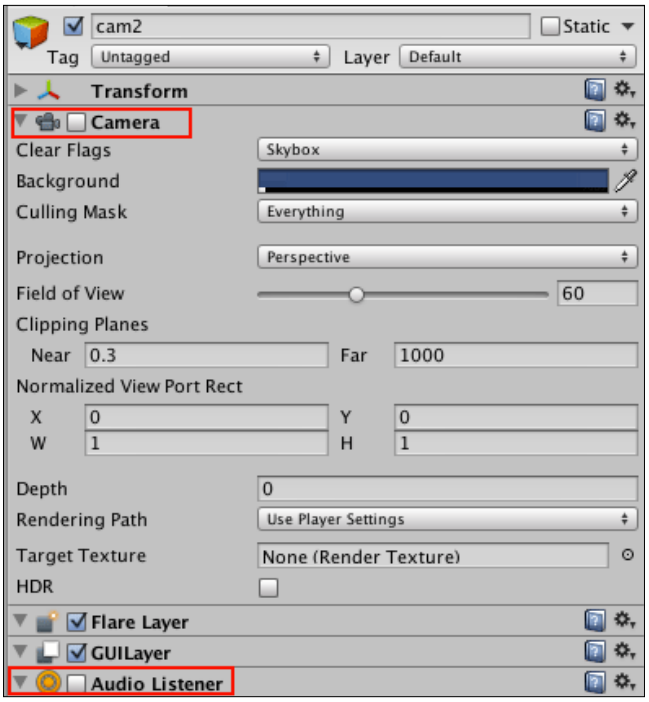
In order to follow this recipe, we have prepared a package containing a basic level named `basicScene`. The package is in the folder `0423_02_01_02`.

How to do it...

To implement switchable cameras, follow these steps:

1. Import the `basicLevel` package into your Unity project.
2. In the **Project** view, open `basicScene` from the `02_01_02` folder. This is a basic scene featuring a directional light, a camera, and some geometry.
3. Add two more cameras to the scene. You can do it through the **Create** drop-down menu on top of the **Hierarchy** view. Rename them `cam1` and `cam2`.
4. Change the `cam2` camera's position and rotation so it won't be identical to `cam1`.
5. Create an **Empty** game object by navigating to **Game Object** | **Create Empty**. Then, rename it `Switchboard`.
6. In the **Inspector** view, disable the **Camera** and **Audio Listener** components of both `cam1` and `cam2`.

www.it-ebooks.info



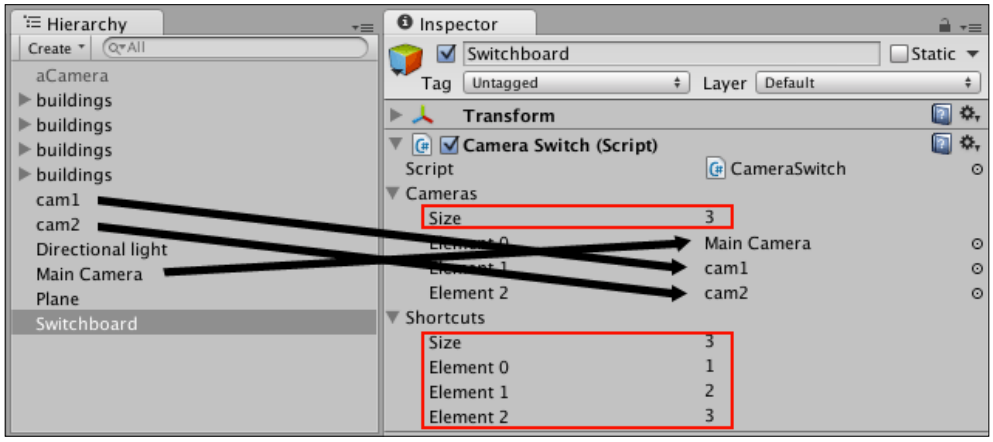
7. In the **Project** view, create a new C# script. Rename it `CameraSwitch` and open it in your editor.
8. Open your script and replace everything with the following code:

```
using UnityEngine;

public class CameraSwitch : MonoBehaviour {
    public GameObject[] cameras;
    public string[] shortcuts;
    public bool  changeAudioListener = true;
    void  Update (){
           int i = 0;
           for(i=0; i<cameras.Length; i++){
                  if (Input.GetKeyUp(shortcuts[i]))
                  SwitchCamera(i);
        }
    }

    void  SwitchCamera ( int index  ){
              int i = 0;
           for(i=0; i<cameras.Length; i++){
```


www.it-ebooks.info

```
                    if(i != index){
                        if(changeAudioListener){
                            cameras[i].
GetComponent<AudioListener>().enabled = false;
                        }
                        cameras[i].camera.enabled = false;
                    } else {
                        if(changeAudioListener){
                            cameras[i].
GetComponent<AudioListener>().enabled = true;
                        }
                        cameras[i].camera.enabled = true;
                    }
                }
            }
        }
```

9. Attach `CameraSwitch` to the `Switchboard` game object.
10. In the **Inspector** view, set both **Cameras** and **Shortcuts** size to **3**. Then, drag the scene cameras into the **Cameras** slots, and type `1`, `2`, and `3` into the **Shortcuts** text fields, as shown in the next screenshot.



11. Play your scene and test your cameras.

How it works...

The script is very straightforward. All it does is capture the key pressed and enable its respective camera (and its **Audio Listener**, in case the **Change Audio Listener** option is checked).

www.it-ebooks.info

There's more...

Here are some ideas on how you could try twisting this recipe a bit.

Using a single-enabled camera

A different approach to the problem would be keeping all the secondary cameras disabled and assigning their position and rotation to the main camera via a script (you would need to make a copy of the main camera and add it to the list, in case you wanted to save its transform settings).

Triggering the switch from other events

Also, you could change your camera from other game object's scripts by using a line of code, such as the one shown here:

```
GameObject.Find("Switchboard").GetComponent("CameraSwitch").
SwitchCamera(1);
```

See also

- The *Making an inspect camera* recipe.

Customizing the lens flare effect

As anyone who has played a game set in an outdoor environment in the last 15 years can tell you, the **lens flare** effect is used to simulate the incidence of bright lights over the player's field of view. Although it has become a bit overused, it is still very much present in all kinds of games. In this recipe, we will create and test our own lens flare texture.

Getting ready

In order to continue with this recipe, it's strongly recommended that you have access to image editor software such as Adobe Photoshop or GIMP. The source for lens texture created in this recipe can be found in the `0423_02_03` folder.

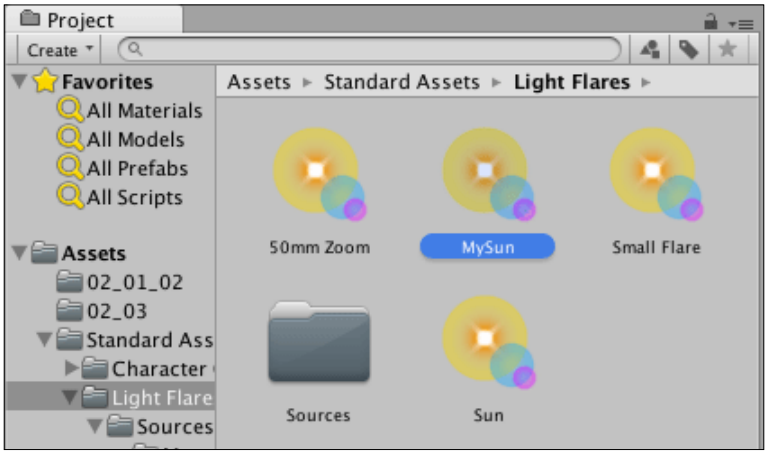
How to do it...

To create a new lens flare texture and apply it to the scene, follow these steps:

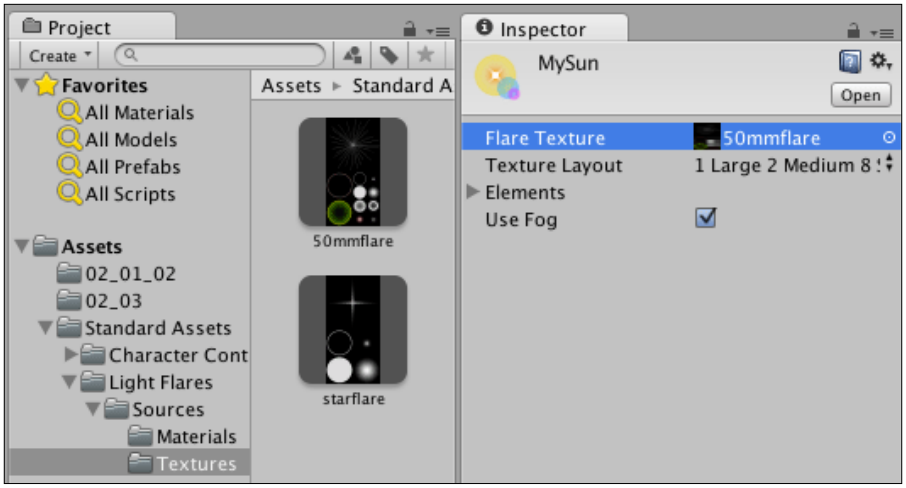
1. Import Unity's **Character Controller** package by navigating to **Assets** | **Import Package** | **Character Controller**.
2. Do the same for the **Light Flares** package.

www.it-ebooks.info

3. In the **Hierarchy** view, use the **Create** button to add a **Directional Light** effect to your scene.
4. Select your camera and add a **Mouse Look** component by accessing the **Component | Camera Control | Mouse Look** menu option.
5. In the **Project** view, locate the **Sun** flare (inside **Standard Assets | Light Flares**), duplicate it and rename it to **MySun**, as shown in the following screenshot:

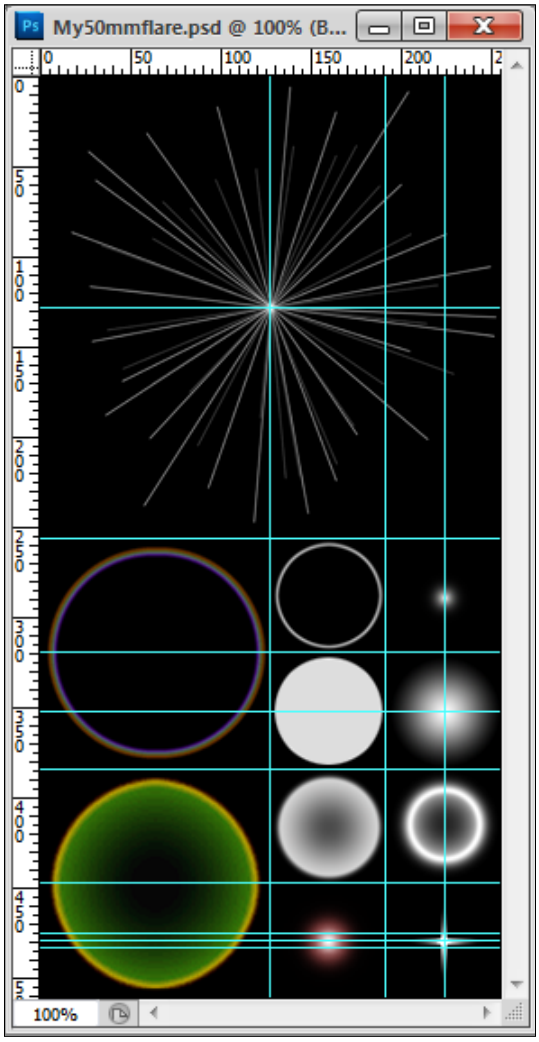


6. In the **Inspector** view, click **Flare Texture** to reveal the base texture's location in the **Project** view. It should be a texture named **50mmflare**.
7. Duplicate the texture and rename it `My50mmflare`.



www.it-ebooks.info

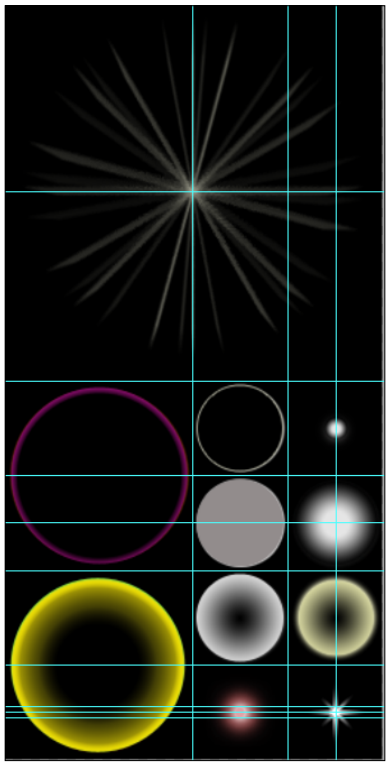
8. Right-click **My50mmflare** and choose **Open**. This should open the file (actually a `.psd`) in your image editor. If you're using Adobe Photoshop, you might see the guidelines for the texture, as shown here:



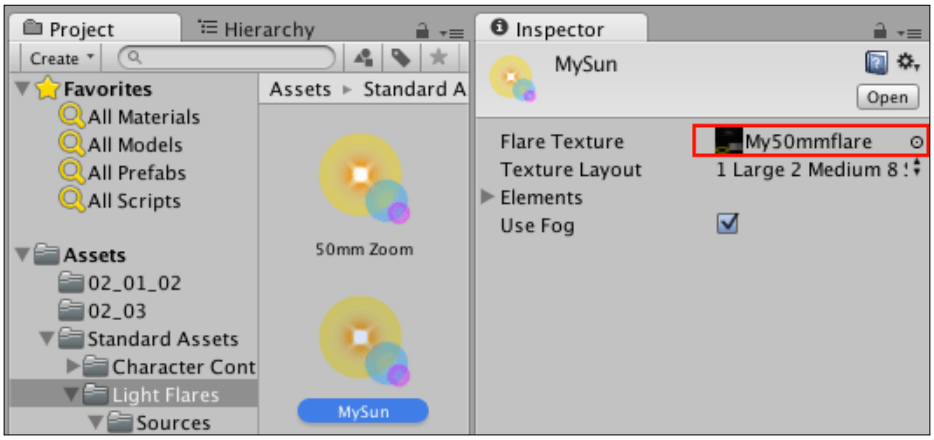
9. To create the light rings, create new **Circle** shapes and add different **Layer Effects** such as **Gradient Overlay**, **Stroke**, **Inner Glow**, and **Outer Glow**.

www.it-ebooks.info

10. Recreate the star-shaped flares by editing the originals or by drawing lines and blurring them.

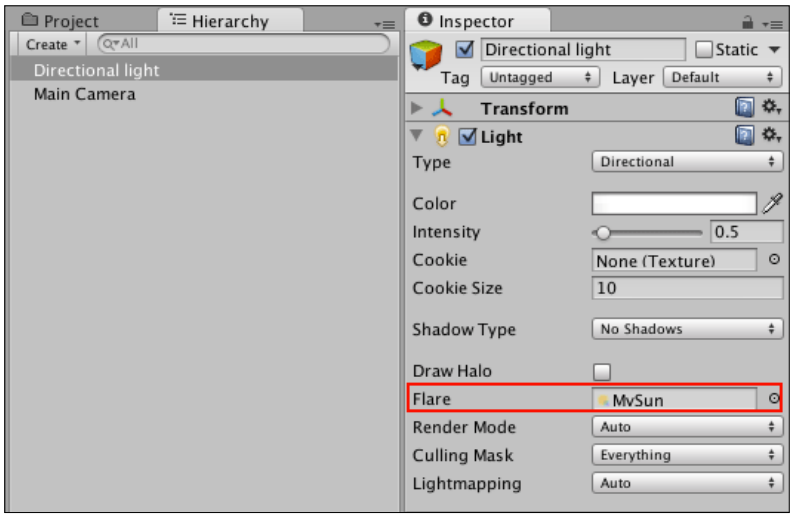


11. Save the file and go back to the Unity Editor.
12. In the **Inspector** view, select **MySun**, and set **Flare Texture** to **My50mmflare**:



www.it-ebooks.info

13. Select **Directional Light** and, in the **Inspector** view, set **Flare** to **MySun**.



14. Play the scene and move your mouse around. You will be able to see the lens flare as the camera faces the light.

How it works...

We have used Unity's built-in lens flare texture as a blueprint for our own. Once applied, the lens flare texture will be displayed when the player looks into the approximate direction of the light.

There's more...

Flare textures can use different layouts and parameters for each element. In case you want to learn more about the **Lens Flare** effect, check out Unity's documentation at `http://docs.unity3d.com/Documentation/Components/class-LensFlare.html`.

Making textures from screen content

If you want your game or player to take in-game snapshots and apply it as a texture, this recipe will show you how. This can be very useful if you plan to implement an in-game photo gallery or display a snapshot of a past key moment at the end of a level (Race Games and Stunt Sims use this feature a lot).

www.it-ebooks.info

Getting ready

In order to follow this recipe, please import the `basicTerrain` package, available in the `0423_02_04_05` folder, into your project. The package includes a basic terrain and a camera that can be rotated via a mouse.

How to do it...

To create textures from screen content, follow these steps:

1. Import the Unity package and open the `02_04_05` scene.
2. We need to create a script. In the **Project** view, click on the **Create** drop-down menu and choose **C# Script**. Rename it `ScreenTexture` and open it in your editor.
3. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class ScreenTexture : MonoBehaviour {
    public int photoWidth = 50;
    public int photoHeight = 50;
    public int thumbProportion = 25;
    public Color borderColor = Color.white;
    public int borderWidth = 2;
    private Texture2D texture;
    private Texture2D border;
    private int screenWidth;
    private int screenHeight;
    private int frameWidth;
    private int frameHeight;
    private bool  shoot = false;

    void  Start (){
        screenWidth = Screen.width;
        screenHeight = Screen.height;
        frameWidth = Mathf.RoundToInt(screenWidth * photoWidth *
0.01f);
        frameHeight = Mathf.RoundToInt(screenHeight * photoHeight
* 0.01f);
        texture = new Texture2D (frameWidth,frameHeight,TextureFor
mat.RGB24,false);
        border = new Texture2D (1,1,TextureFormat.ARGB32, false);
        border.SetPixel(0,0,borderColor);
        border.Apply();
    }
```

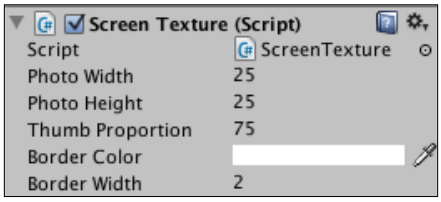

www.it-ebooks.info

```
    void  Update (){
        if (Input.GetKeyUp(KeyCode.Mouse0))
                    StartCoroutine(CaptureScreen());
    }

    void  OnGUI (){
        GUI.DrawTexture(new Rect((screenWidth*0.5f)-
(frameWidth*0.5f) - borderWidth*2,((screenHeight*0.5f)-
(frameHeight*0.5f)) - borderWidth,frameWidth + borderWidth*2,borde
rWidth),border,ScaleMode.StretchToFill);
        GUI.DrawTexture(new Rect((screenWidth*0.5f)-
(frameWidth*0.5f) - borderWidth*2,(screenHeight*0.5f)+(frameHeigh
t*0.5f),frameWidth + borderWidth*2,borderWidth),border,ScaleMode.
StretchToFill);
        GUI.DrawTexture(new Rect((screenWidth*0.5f)-
(frameWidth*0.5f)- borderWidth*2,(screenHeight*0.5f)-(frameHeight*
0.5f),borderWidth,frameHeight),border,ScaleMode.StretchToFill);
        GUI.DrawTexture(new Rect((screenWidth*0.5f)+(frameWidth*0.
5f),(screenHeight*0.5f)-(frameHeight*0.5f),borderWidth,frameHeight
),border,ScaleMode.StretchToFill);
        if(shoot){
            GUI.DrawTexture(new Rect (10,10,frameWidth*thumbPropo
rtion*0.01f,frameHeight*thumbProportion* 0.01f),texture,ScaleMode.
StretchToFill);
        }
    }

    IEnumerator  CaptureScreen (){
        yield return new WaitForEndOfFrame();
        texture.ReadPixels(new Rect((screenWidth*0.5f)-(frameWidth
*0.5f),(screenHeight*0.5f)-(frameHeight*0.5f),frameWidth,frameHeig
ht),0,0);
        texture.Apply();
        shoot = true;
    }
}
```

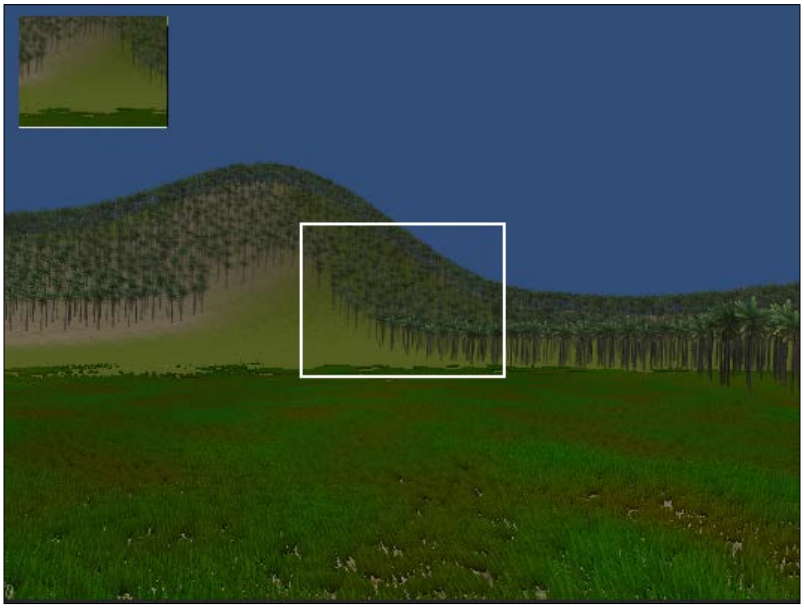
4. Save your script and apply it to the **Main Camera** game object.

www.it-ebooks.info

5. In the **Inspector** view, change the values for the **Screen Capture** component, leaving **Photo Width** and **Photo Height** as **25** and **Thumb Proportion** as **75**, as shown here:



6. Play the scene. You will be able to take a snapshot of the screen (and have it displayed on the top-left corner) by clicking the mouse button.



How it works...

Clicking the mouse triggers a function that reads pixels within the specified rectangle and applies them into a texture that is drawn by the GUI.

There's more...

Apart from displaying the texture as a GUI element, you could use it in other ways.

www.it-ebooks.info

Applying your texture to a material

You apply your texture to an existing object's material by adding a line similar to `GameObject.Find("MyObject").renderer.material.mainTexture = texture;` at the end of the `CaptureScreen` function.

Using your texture as a screenshot

You can encode your texture as a PNG image file and save it. Check out Unity's documentation on this feature at `http://docs.unity3d.com/Documentation/ScriptReference/Texture2D.EncodeToPNG.html`.

Zooming a telescopic camera

In this recipe, we will create a telescopic camera that zooms in whenever the left mouse button is pressed. This can be very useful, for instance, if we have a sniper in our game.

Getting ready

In order to follow this recipe, please import the `basicTerrain` package, available in the `0423_02_04_05` folder, into your project. The package includes a basic terrain and a camera that can be rotated via a mouse.

How to do it...

To create a telescopic camera, follow these steps:

1. Import the Unity package and open the `02_04_05` scene.
2. We need to create a script. In the **Project** view, click on the **Create** drop-down menu and choose **C# Script**. Rename it `TelescopicView` and open it in your editor.
3. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class TelescopicView : MonoBehaviour{
    public float ZoomLevel = 2.0f;
    public float ZoomInSpeed = 100.0f;
    public float ZoomOutSpeed = 100.0f;
    private float initFOV;
    //private Vignetting vignette;
    //public float vignetteAmount = 10.0f;
```


www.it-ebooks.info

```
    void Start(){
        initFOV = Camera.main.fieldOfView;
             //vignette = this.GetComponent("Vignetting") as
Vignetting;
    }
    void Update(){
        if (Input.GetKey(KeyCode.Mouse0)){
            ZoomView();
        }else{
            ZoomOut();
        }
    }
    void ZoomView(){
        if (Mathf.Abs(Camera.main.fieldOfView - (initFOV /
ZoomLevel)) < 0.5f){
            Camera.main.fieldOfView = initFOV / ZoomLevel;
                    // vignette. intensity = vignetteAmount;
        }else if (Camera.main.fieldOfView - (Time.deltaTime *
ZoomInSpeed) >= (initFOV / ZoomLevel)){
            Camera.main.fieldOfView -= (Time.deltaTime *
ZoomInSpeed);
           // vignette. intensity = vignetteAmount * (Camera.main.
fieldOfView - initFOV)/((initFOV / ZoomLevel) - initFOV);
        }
    }
    void ZoomOut(){
        if (Mathf.Abs(Camera.main.fieldOfView - initFOV) < 0.5f){
            Camera.main.fieldOfView = initFOV;
           // vignette. intensity = 0;
        }else if (Camera.main.fieldOfView + (Time.deltaTime *
ZoomOutSpeed) <= initFOV){
            Camera.main.fieldOfView += (Time.deltaTime *
ZoomOutSpeed);
            //vignette.intensity = vignetteAmount * (Camera.main.
fieldOfView - initFOV)/((initFOV / ZoomLevel) - initFOV);
        }
    }
}
```

4. Save your script and apply it to the **Main Camera** game object.
5. Play your scene. You should be able to zoom in the camera view by clicking the mouse button and zoom out by releasing it.

www.it-ebooks.info

6. The next steps are for only those using Unity Pro. If you are using Unity Pro, please stop the scene.
7. Import Unity's **Image Effects** package by navigating to **Assets** | **Import Package** | **Image Effects (Pro Only)**.
8. Select the **Main Camera** option and apply the **Vignette** image effect (by navigating to **Component** | **Image Effects** | **Vignette**).
9. Open the `TelescopicView` script and uncomment every commented line (these are the ones referencing the `vignetteAmount` variable).
10. Save the script and play the level. You should see an animated vignetting effect in addition to the zooming:

How it works...

The zooming effect is actually caused by the increment made by the script on the value of the camera's **Field of View** property, while the mouse button is pressed.

There's more...

If you are using Unity Pro, you could also add a variable to control the **Blur Vignette** level of the **Vignette** image effect.

www.it-ebooks.info

Making an inspect camera

Inspect cameras are a very popular way of displaying products online. These virtual displays usually feature a camera that orbits around and zooms in and out the product. In this recipe, we will learn how to implement such a camera using a standard Unity component as a starting point.

Getting ready

In order to follow this recipe, please import the `inspectScene` package, available in the `0423_02_06` folder, into your project. The package includes a basic scene containing the 3D model of a mobile phone and a camera.

Please note this recipe involves editing a JavaScript file provided by Unity. In case you want the C# version of the final script, we have included it in the 0423_02_06 folder.

How to do it...

To create an inspect camera, follow these steps:

1. Import the `inspectScene` Unity package and open the scene available in the `02_06` folder, also named `inspectScene`.
2. Add a directional light to the scene, making it cast light in the same direction the camera is facing.
3. In the **Hierarchy** view, drag the **Directional Light** into the **Main Camera** game object, making it a child of the **Main Camera** game object.
4. Make sure the `MouseOrbit` script is available at the **Project** view (it should be inside the **Standard Assets** | **Scripts** | **Camera Scripts** folder). If not, import the **Scripts** package by navigating to **Assets** | **Import Package** | **Scripts**.
5. Duplicate the `MouseOrbit` script and rename the new copy to `InspectCamera`.
6. Open the `InspectCamera` script in your editor.
7. Insert, before the line starting with `@script`, insert the following code:

```
var zoomInLimit = 2.0;
var zoomOutLimit = 1.0;
private var initialFOV : float;
```

8. Then, change the line starting with `@script` to:

```
@script AddComponentMenu("Camera-Control/Inspect Camera")
```


www.it-ebooks.info

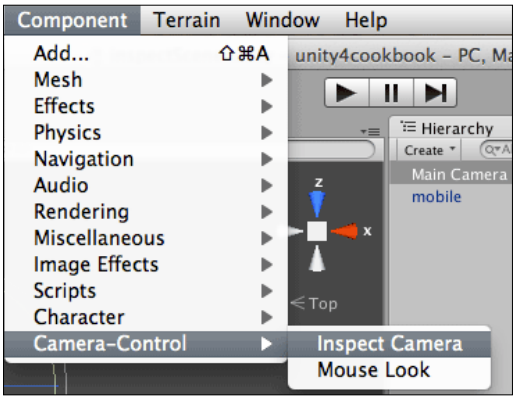
9. Also, add the following lines of code right after the `Start` function begins:

```
initialFOV = camera.fieldOfView;
transform.position = new Vector3(0.0f, 0.0f, -distance) + target.
position;
```

10. Then, replace all of the `LateUpdate` function with the following code:

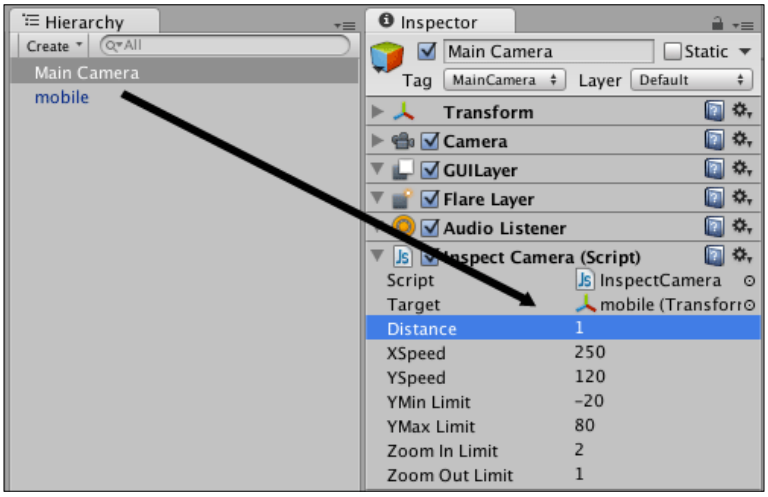
```
function LateUpdate () {
   if (target && Input.GetMouseButton(0)) {
            if(Input.GetKey(KeyCode.RightShift) || Input.
GetKey(KeyCode.LeftShift)){
                  var zoom = camera.fieldOfView - Input.GetAxis
("Mouse Y");
                  if(zoom >= initialFOV / zoomInLimit && zoom <=
initialFOV / zoomOutLimit){
                        camera.fieldOfView -= Input.GetAxis
("Mouse Y");
                  }
             } else {
                  x += Input.GetAxis("Mouse X") * xSpeed * 0.02;
            y -= Input.GetAxis("Mouse Y") * ySpeed * 0.02;
            }
     y = ClampAngle(y, yMinLimit, yMaxLimit);
     var rotation = Quaternion.Euler(y, x, 0);
     var position = rotation * Vector3(0.0, 0.0, -distance) +
target.position;
       transform.rotation = rotation;
       transform.position = position;
    }
}
```

11. Save the script. Then, select the **Main Camera** option in the **Hierarchy** view and add the **Inspect Camera** component by navigating to **Component** | **Camera-Control** | **Inspect Camera**, as shown in the following screenshot:



www.it-ebooks.info

12. The **Inspect Camera** component should now be available in the **Inspector** window for the **Main Camera**. From the **Hierarchy** view, drag the **mobile** game object into the **Target** slot. Then, change the **Distance** value to **1**:



13. Play the scene. Move the mouse and keep the left button pressed to look around the object. To zoom in and out, keep the left button and the *Shift* key pressed, and move the mouse vertically.

How it works...

Unlike Unity's `MouseOrbit`, our `InspectCamera` script checks if the mouse button is pressed as a condition to move the camera. We have also implemented a zooming feature that changes the value of the camera's **Field of View** when the *Shift* key and a mouse button are pressed, and the mouse is moved vertically.

There's more...

If you want a C# version of the script, don't worry; we have included it in the `0423_02_07` folder. Also, should you need to use another key instead of *Shift*, check out Unity's documentation for the appropriate key codes at `http://docs.unity3d.com/Documentation/ScriptReference/KeyCode.html`.

Creating particle effects using Shuriken

From Unity 3.5 onwards, much more impressive effects can be created using particle systems. Many effects which used to need scripting can be achieved entirely through the Shuriken particle system settings.

www.it-ebooks.info

Getting ready

It is useful to review important properties and the parameter value selection methods of Unity particle systems before actually trying to work with them in the editor itself.

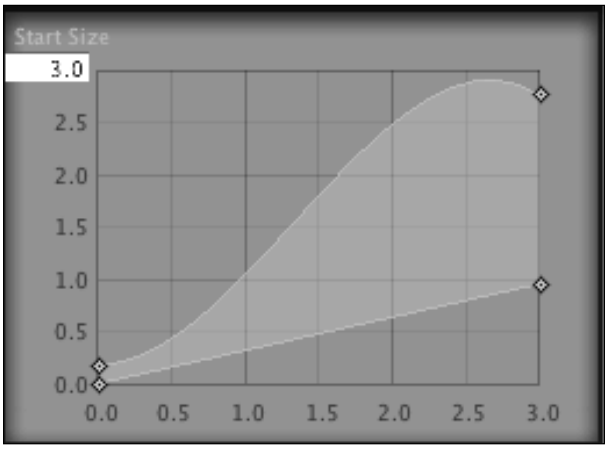
Some of the fundamental properties of particle systems include the following:

- **Energy**: How long a particle will "live" until it is destroyed
- **Looping**: Whether the entire population of particles will be recreated again once the last particle has "died"
- **Speed, direction, and rotation**: Each particle has its own transform, and when a new particle is created the particle system determines (from the settings you have chosen) that particle's direction and speed of movement and any rotation

In addition, it is important to understand the different methods that values are chosen for particle properties:

- **Constant**: The same value for all of the particles
- **Curve**: How the values will change over the duration of each cycle of the particle system (for example, particles start small and get larger over time)
- **Random number between maximum and minimum constants**: Values are randomly chosen from the same range over each cycle of the particle system
- **Random number between two curves**: This allows the range of values from which values are chosen to change over the duration of each cycle of the particle system

  The following screenshot shows two particle-size curves, that ensures particles start small (between `0.0` and `0.2`), and get larger over the 3 second (x-axis) cycle length. However, the size of particles is random between the two curves (a value always chosen in the "red" zone), so towards the end of the cycle, particles will be a random size between `1.0` and `2.8`:



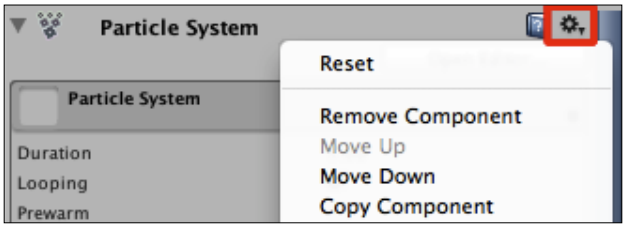
www.it-ebooks.info

How to do it...

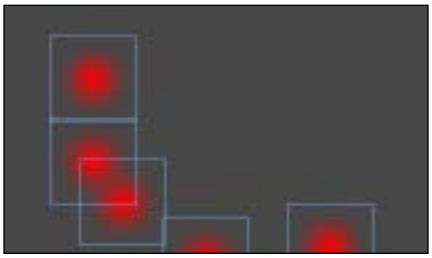
To create a particle effects using Shuriken, follow these steps:

1. Create a new Unity project and import the **Particles** Unity package.
2. Create a new particle system.
3. Reset the particle system to its defaults by choosing **Reset** from the component's **Context** menu in the **Inspector**.

To view the **Context** menu, either right-click on the component's name, or click the tiny gear icon at the top-right corner of the component's section in the **Inspector** view, as shown here:



4. Switch on (check) **Wireframe** for the **Render** section of the particle system, shown below:



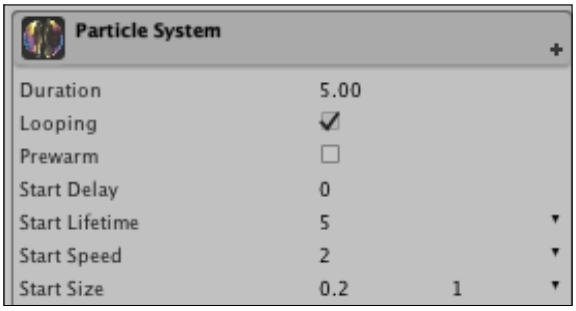
Wireframes can aid in understanding the position and movement of particles by drawing a blue bounding rectangle around each particle in the **Scene** window, and a white bounding rectangle around the complete set of particles of each frame.

www.it-ebooks.info

5. Display the **Renderer** properties (double-click the darker grey title bar) and change the renderer's **Material** parameter to **SoapBubble**:

You'll find materials from the Unity particles package located in the Project folder as follows: **Standard Assets** | **Particles** | **Sources** | **Materials**.

6. Set **Start Speed** to **2** and set **Start Size** to a random number between the constants **0.2** and **1**:



How it works...

You've created a particle system which emits differently-sized rising particles showing a soap bubble. They move relatively slowly (since you set **Start Speed** to **2**). Each particle shows a soap bubble, since you set the renderer to use a material that had a soap bubble texture image. The bubbles vary in size since you set **Start Size** to be a random number between `0.2` and `1`. They float upwards because the system is rotated by -90 on the **X-axis** (the default settings you ensured by clicking **Reset**).

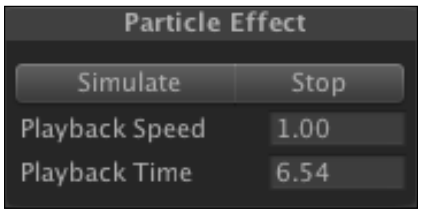
www.it-ebooks.info

There's more...

Here is some information on how to fine-tune and customize this recipe.

Scrubbing back and forward through the particle animation

A nice feature to easily review and ensure that the particle system's behavior is as desired is the ability to "scrub" (move back and forward through the animation). This is achieved by clicking and dragging left or right on **Playback Time** text in the particle **Preview Panel** in the **Scene** window:



Learning more from the Unity documentation

Unity particle systems with Shuriken can be very powerful and therefore complex components of games. The Unity manual provides several pages dedicated to introducing and describing this system. You'll find the `ParticleSystems.html` file in the `/Documentation/Manual/` section of the Unity website, and also in your own computer's Unity application folder.

Adding effects through Particle Effect

Opening the **Window** | **Particle Effect** menu and clicking the plus symbol adds additional particle sub-systems. Complex effects can be created through particle systems working at different times with different settings.

Displaying a mini-map

In many games, a broader view of the scene can be invaluable for navigation and information. Mini-maps are great for giving players that extra perspective they might need when in first- or third-person mode.

Getting ready

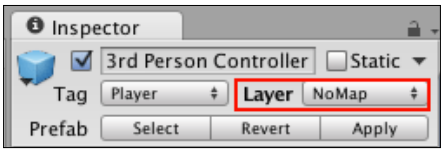
In order to follow this recipe, please import the `miniMap` package, available in the `0423_02_08` folder, into your project. The package includes a scene containing a third-person controller inside a simple maze, and also a texture named `marker`.

www.it-ebooks.info

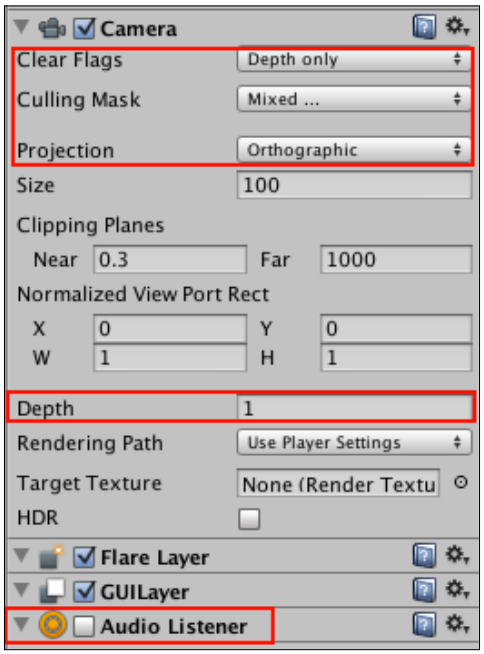
How to do it...

To create a mini-map, follow these steps:

1. Import the `miniMap` Unity package and open the scene named `thirdPerson`, available in the `02_08` folder.
2. Select the **3rd Person Controller** option and, in the **Inspector** view, add it to a new layer named **NoMap**:



3. Create a new camera. Rename it **Map Camera** and, in the **Inspector** view, change its parameters as follows:
   - **Clear Flags**: **Depth Only**
   - **Culling Mask**: **Mixed...** (unselect **NoMap**)
   - **Projection**: **Orthographic**
   - **Depth**: **1** (also, uncheck the camera's **Audio Listener** component)



www.it-ebooks.info

4. In the **Project** window, create a new **C# Script** and name it `GenerateMap`. Open it and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class GenerateMap : MonoBehaviour
{
   public Transform target;
   public Texture2D marker;
   public float camHeight = 1.0f;
   public bool freezeRotation = true;
   public float camDistance = 2.0f;
   public enum ha {left, center, right};
   public enum va {top, middle, bottom};
   public ha horizontalAlignment = ha.left;
   public va verticalAlignment = va.top;
   public enum sd {pixels, screen_percentage};
   public sd dimensionsIn = sd.pixels;
   public int width = 50;
   public int heigth = 50;
   public float xOffset = 0f;
   public float yOffset = 0f;

    void Start(){
        Vector3 angles = transform.eulerAngles;
        angles.x = 90;
        angles.y = target.transform.eulerAngles.y;
        transform.eulerAngles = angles;
        Draw();
    }

    void Update(){
        transform.position = new Vector3(target.transform.
position.x, target.transform.position.y + camHeight, target.
transform.position.z);
        camera.orthographicSize = camDistance;
        if (freezeRotation){
            Vector3 angles = transform.eulerAngles;
            angles.y = target.transform.eulerAngles.y;
            transform.eulerAngles = angles;
        }
    }

    void Draw()    {
```

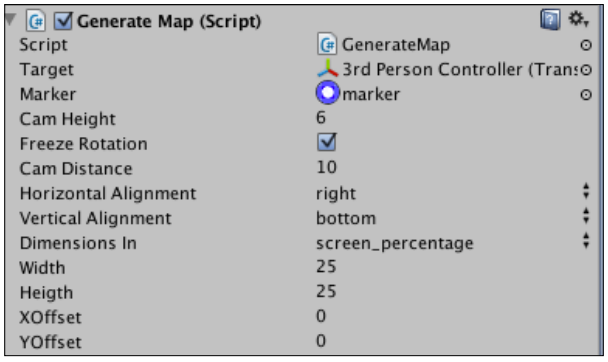

www.it-ebooks.info

```
        int hsize = Mathf.RoundToInt(width * 0.01f * Screen.
width);
            int vsize = Mathf.RoundToInt(heigth * 0.01f * Screen.
height);
            int hloc = Mathf.RoundToInt(xOffset * 0.01f * Screen.
width);;
            int vloc = Mathf.RoundToInt((Screen.height - vsize) -
(yOffset * 0.01f * Screen.height));
            if(dimensionsIn == sd.screen_percentage){
               hsize = Mathf.RoundToInt(width * 0.01f * Screen.
width);
               vsize = Mathf.RoundToInt(heigth * 0.01f * Screen.
height);
            } else {
                  hsize = width;
                  vsize = heigth;
            }

               switch(horizontalAlignment){
                   case ha.left:
                         hloc = Mathf.RoundToInt(xOffset * 0.01f *
Screen.width);
                   break;
                   case ha.right:
                         hloc = Mathf.RoundToInt((Screen.width -
hsize) - (xOffset * 0.01f * Screen.width));
                      break;
                   case ha.center:
                         hloc = Mathf.RoundToInt(((Screen.width *
0.5f) - (hsize * 0.5f)) - (xOffset * 0.01f * Screen.height));
                      break;
               }
               switch(verticalAlignment){
                      case va.top:
                          vloc = Mathf.RoundToInt((Screen.height -
vsize) - (yOffset * 0.01f * Screen.height));
                      break;
                   case va.bottom:
                         vloc = Mathf.RoundToInt(yOffset * 0.01f *
Screen.height);
                      break;
                   case va.middle:
                         vloc = Mathf.RoundToInt(((Screen.height *
0.5f) - (vsize * 0.5f)) - (yOffset * 0.01f * Screen.height));
                      break;
               }
```


www.it-ebooks.info

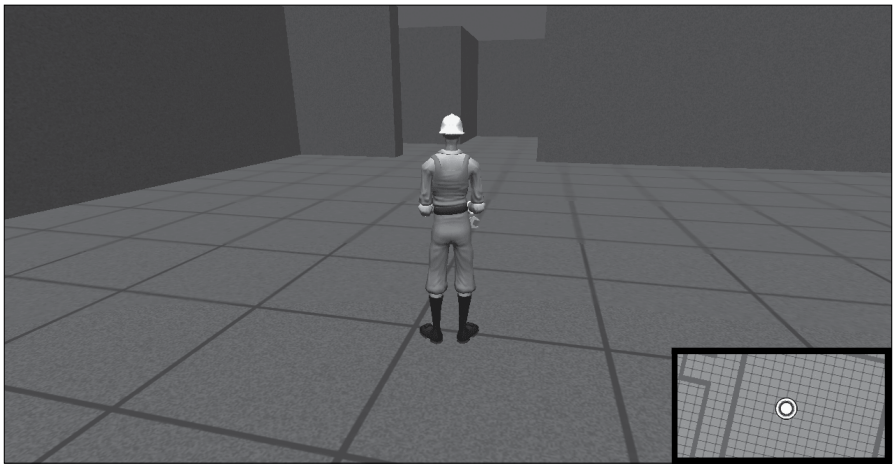
```
            camera.pixelRect = new Rect(hloc,vloc,hsize,vsize);
        }
    void OnGUI(){
            Vector3 markerPos = camera. camera.WorldToViewportPoint
(target.position);
            int pointX =  Mathf.RoundToInt((camera.pixelRect.xMin +
camera.pixelRect.xMax) * markerPos.x);
            int pointY =  Mathf.RoundToInt(Screen.height - (camera.
pixelRect.yMin + camera.pixelRect.yMax) * markerPos.y);
            GUI.DrawTexture( new Rect(pointX-(marker.width *
0.5f),pointY-(marker.height * 0.5f),marker.width,marker.height),
marker, ScaleMode.StretchToFill, true, 10.0f);
        }
}
```

5. Save the script and attach it to the **Map Camera**. Then, in the **Inspector** view, change parameters of the **Generate Map** component as follows:
    - **Target**: **3rd Person Controller**
    - **Marker**: **marker**
    - **Cam Height**: **6**
    - **Cam Distance**: **10**
    - **Horizontal Alignment**: **Right**
    - **Vertical Alignment**: **Bottom**
    - **Dimensions In**: **screen_percentage**
    - **XOffset** and **YOffset**: **0**
    - **Width**: **25**
    - **Height**: **25**



6. Play the scene. You should be able to see the mini-map functioning in the bottom-right corner of the screen:

www.it-ebooks.info



How it works...

Our script performs two main actions: it sets up the viewport according to the desired position and dimensions, and also adjusts the camera's transform settings, making it follow the main character from a top-view perspective. Plus, in order to make the map look cleaner, it hides our main character while displaying a marker where he should be.

There's more...

If you want to experiment more with your mini-map, read on.

Using Render Texture (Pro only)

If you use Unity Pro, you could always use **Render Texture** and display your map using **GUI.DrawTexture** or **Graphics.DrawTexture**. For more information on Render Texture, please check the Unity documentation at `http://docs.unity3d.com/Documentation/Components/class-RenderTexture.html`.

Adapting your mini-map to other styles

You could easily modify this recipe to make a top or isometric view of a racing game circuit map. Just position the camera manually and prevent it from following the character. Also, don't forget to add markers for all the vehicles!

See also

- The *Creating a picture-in-picture effect* recipe.

www.it-ebooks.info

www.it-ebooks.info

3
Creating Maps and Materials

In this chapter, we will cover:

- Creating a reflective material
- Creating a self-illuminated material
- Creating specular texture maps
- Creating transparency texture maps
- Using cookie textures to simulate a cloudy outdoor
- Creating a color selection dialog
- Combining textures in real time through the GUI
- Highlighting materials at mouse over
- Animating textures by looping through array of materials (e.g. simulated video)
- Disabling culling for a material

Introduction

Most materials are created with the help of texture maps. To create those maps, which are actually bitmap files (regardless of their file extension), it is necessary to use an image editor such as Adobe Photoshop (which is the industry standard and has its native format supported by Unity), GIMP, and so on. In order to follow the recipes in this chapter, it's strongly recommended that you have access to software like this.

www.it-ebooks.info

When saving texture maps, specially the ones that have an alpha channel, you might want to choose an adequate file format. PSD, Photoshop's native format, is practical for preserving the original artwork in many layers. TGA is also a good option for preserving the alpha channel—although the image will be flattened.

Finally, before you start, it might be a good idea to read Unity's documentation on textures. It can be found online at `http://unity3d.com/support/documentation/Manual/Textures.html`.

Creating a reflective material

Metal, car paint, and glossy plastic surfaces are just some very recurrent examples of materials that need reflectiveness. Luckily for us, Unity includes reflective shaders that, when properly configured, can help us to achieve the look that we are after.

Getting ready

For this recipe, we will prepare two texture maps: the **Base** map and the **Reflection Cubemap**. The Base map will be a RGBA image, where the alpha channel will assign the reflection level from completely opaque (black) to completely reflective (white). The Reflection Cubemap can be made from either six or, as shown in this recipe, a single texture.

To help us focus on this recipe's main subject, which is setting up a reflective material, a Unity package containing a previously made scene, and also a `.jpg` file to be used as reflection map, are available inside the `0423_03_01` folder.

How to do it...

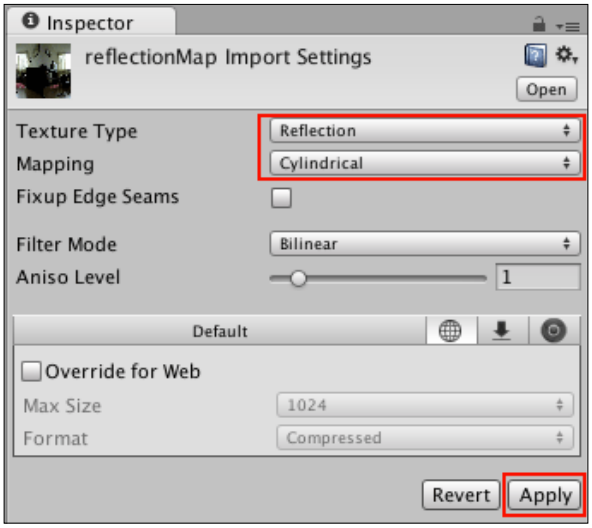
To create a reflective material, follow these steps:

1. Import the `reflectiveMaterial` Unity package.
2. Open the `reflectiveMaterialLevel` level. There will be a single 3D object (a battery) that rotates around its own axis when the level is played. We can also orbit the camera around it.

www.it-ebooks.info

3. Stop the level from running and, in the **Hierarchy** view, select the battery. The **Inspector** view will display its material. Use the drop-down menu to set **Shader** from **Diffuse** to **Reflective/Specular**:

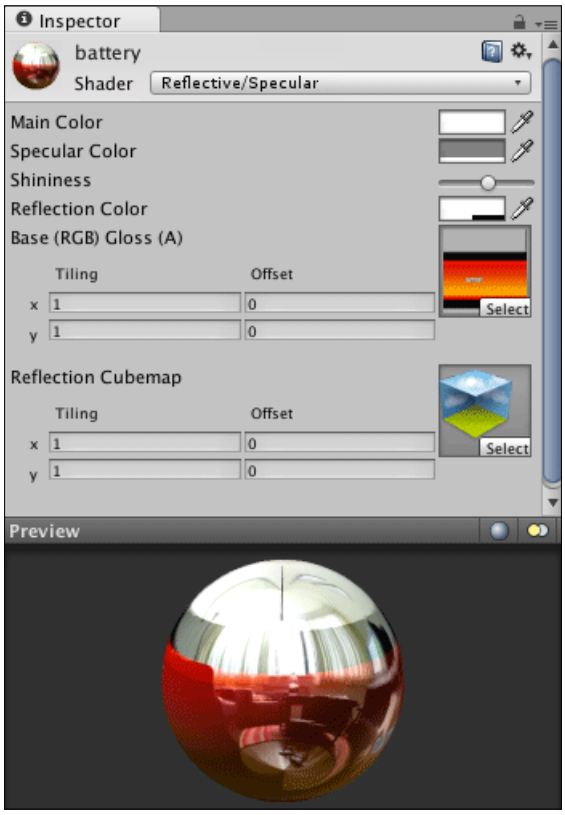
4. Now we must build our Reflection Cubemap. In the **Project** view, select the **reflectionMap** texture. The **Texture Importer** settings will be displayed in the **Inspector** view. Using the **Texture Type** drop-down menu, choose **Reflection**. Then, change the **Mapping** option to **Cylindrical**. Finally, click **Apply** to apply the changes we've made, as shown in the following screenshot:



5. Select the `reflectionMap` asset in your **Project** view. It has changed from a 2D texture to a Cubemap (as you can see from its icon).

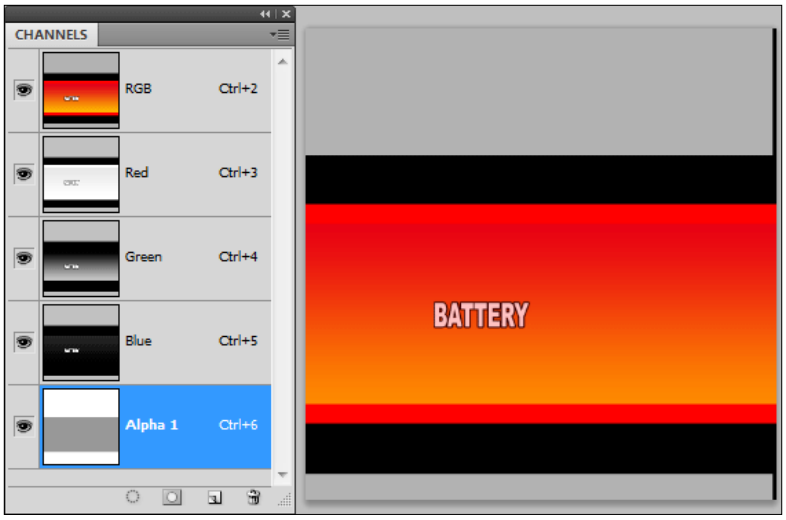
www.it-ebooks.info

6. Select the battery material and apply the `reflectionMap` asset into the **Reflection Cubemap** slot:



7. Play the level. The reflection effect is visible, but it could be smoother in some areas of the texture.
8. Open the base texture (named `batteryBase`) in your image editor.
9. In your image editor, make all the channels visible and select the **Alpha** channel window (in Photoshop this can be done by navigating to **Window** | **Channels**).
10. The Alpha channel is completely white. Paint a grey rectangle in the region correspondent to the battery's label. In this example, we have used a grey color of **Red**, and also used **Green** and **Blue** values of `152`:

www.it-ebooks.info



11. Save the file to make the changes and wait for Unity to reload the image. Reflection should now be more discreet inside the label area.

How it works...

The base texture map's alpha channel sets the reflectiveness and, in this example, the specular level of the material according to each pixel's brightness. That means the Reflection Cubemap texture will be more visible where the alpha channel is white.

The Reflection Cubemap is actually made from six different images, which were generated from our original image file by the **Texture Importer** settings when we set **Texture Type** to **Reflection**. These six images are mapped as top, bottom, front, back, left, and right.

There's more...

Reflection Cubemaps can be done in many ways and have different mapping properties.

Using six different texture maps

Instead of using a single texture map, you could also create an empty Reflection Cubemap in the **Project** view and assign six different image files illustrating the top, bottom, front, back, left, and right of your environment.

www.it-ebooks.info

Experimenting with mapping and shader options

The cylindrical mapping we used was the best choice for the battery model. Experiment with other settings for different shapes. Besides the Base and Reflection texture maps, you can also experiment with the other shader settings such as **Main**, **Specular** and **Reflection Colors** (that can be used to tint the material), and also **Shininess** (which controls the specular dispersion).

Reflecting actual scene geometry

If you're using Unity Pro, you could make a reflection map from the actual level geometry. In fact, it could either reflect your level in real time or bake a single frame into the Reflection Cubemap. Unity's documentation has all the instructions you need, including the complete script code, at `http://unity3d.com/support/documentation/ScriptReference/Camera.RenderToCubemap.html`.

Creating a self-illuminated material

Self-illuminated materials can be used to simulate a variety of objects, from LED mobile displays to futuristic Tron suits. In this recipe, we will learn how to configure this kind of material and its texture maps.

Getting ready

As we will create an LCD display in this example, please make sure you have a font in that style installed on your computer. If not, you can find several free LCD fonts on specialized websites, such as `www.dafont.com`.

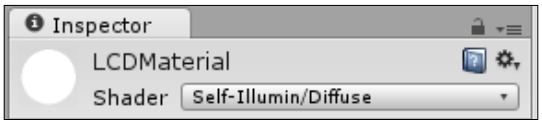
How to do it...

To create a self-illuminated material, follow these steps:

1. Create a new material named `LCDMaterial`.

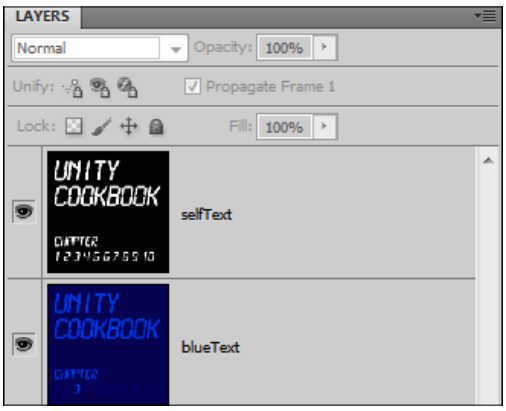
The easiest way to do that is to access the **Project** view, click the **Create** drop-down menu and choose **Material**.

2. Select your material. In the **Inspector** view, under the material's name, use the drop-down menu to change its **Shader** parameter to **Self-Illumin/Diffuse**:

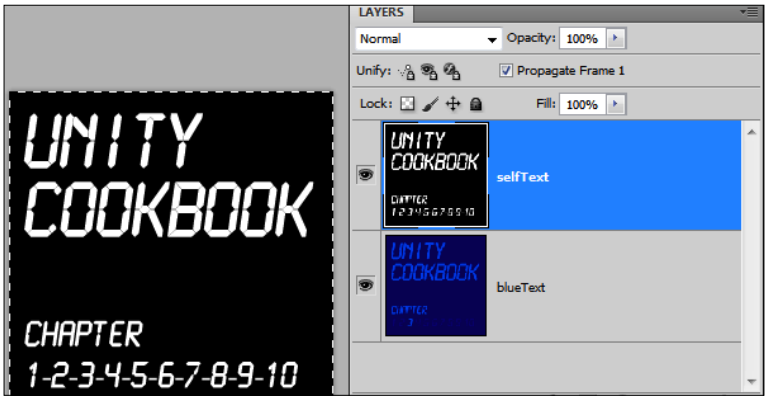


www.it-ebooks.info

3. Open your image editor (we'll use Adobe Photoshop to illustrate the next steps).
4. Create a new image. Let's make it 256 pixel both wide and tall, at 72 dpi (RGB mode).
5. Create a new layer and fill it with a dark blue color (for instance, **R:8**, **G:16**, **B:99**). Name it `blueBg`.
6. Create a type layer and write your LCD text in light blue (**R:8**, **G:90**, **B:231**). Name it `blueText`.
7. Duplicate the `blueBg` and `blueText` layers. Rename them to `selfBg` and `selfText` respectively.
8. Change the color of the `selfText` type to white. Then, fill the `selfBg` layer in black.
9. Merge the `selfText` and `selfBg` layers. Then, merge the `blueText` and `blueBg` layers:

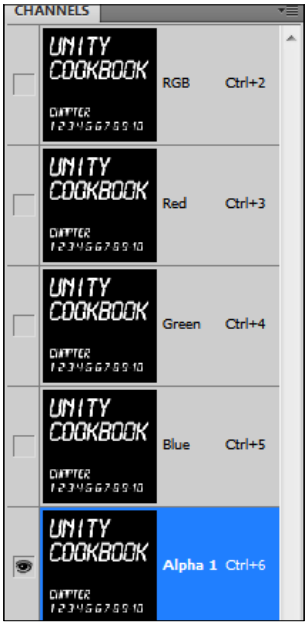


10. Make a selection of the entire black and white layer (*Ctrl* + *A* on Windows, *Command* + *A* on Mac OS). Then, copy it (using *Ctrl* + *C* or *Command* + *C*):



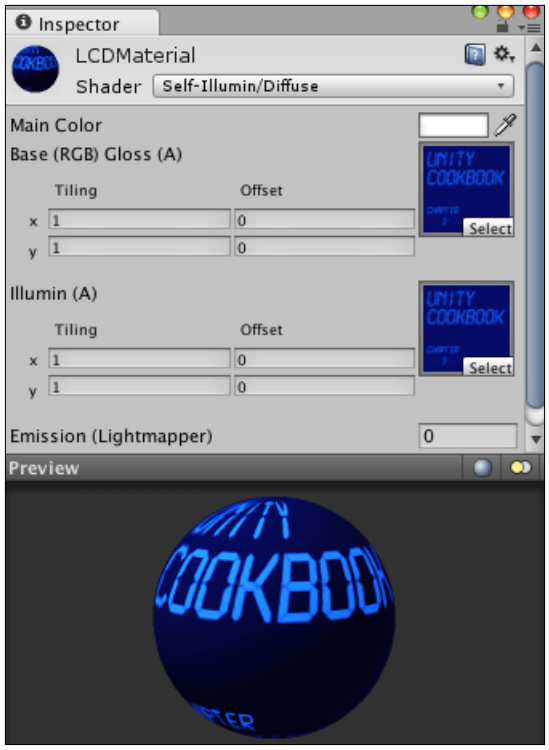
www.it-ebooks.info

11. Open the **Channels** window (in Photoshop this can be done by navigating to **Window** | **Channels**).
12. There should be three channels: **Red**, **Green**, and **Blue**. Create a new channel. That will be the Alpha channel that controls the level of self-illumination and gloss.
13. Paste (*Ctrl* + *V* or *Command* + *V*) your black and white layer into the Alpha channel:



14. In the **Layers** window, hide or delete the merged black and white layer.
15. Save your file (`.psd` or `.tga` formats are good options, as they keep the alpha channel).
16. Inside the Unity Editor, import your image file access through the **Assets** menu, clicking on **Import New Asset...**.
17. In the **Project** view, choose **LCDMaterial**. Then, select your image file as both **Base** and **Illumin** maps (by clicking on the **Select** button or dragging them from the **Project** view to the material slots). Your self-illuminated material is ready, as shown in the following screenshot:

www.it-ebooks.info



How it works...

Unity is able to read four channels of a texture map: R (red), G (green), B (blue), and A (alpha). Self-illuminated shaders use RGB channels as the base texture (often referred to as the diffuse texture), while using the Alpha to illuminate the material according to each pixel's brightness level.

There's more...

Here is some more information on self-illuminated materials.

Using independent maps

Please note that you can use two different image files as the Base and Illumination maps. In this case, the Base texture's alpha channel will be controlling the material's gloss.

Emitting light over other objects

The **Emission (Lightmapper)** field in the **Material** options can be used to simulate the material's light projection over other objects when baking a lightmap.

www.it-ebooks.info

See also

- The *Creating specular texture maps* recipe.
- The *Creating transparency texture maps* recipe.

Creating specular texture maps

Some surfaces can have both glossy and matte areas. In order to achieve that effect, we can use specular maps.

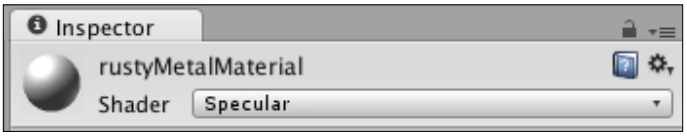
Getting ready

For illustration purposes, we will create a rusted metal material to demonstrate how the specular property can be used to enhance realism. If you don't have a base texture to develop into a specular material, please use the one included in the book's `0423_03_03` content folder.

How to do it...

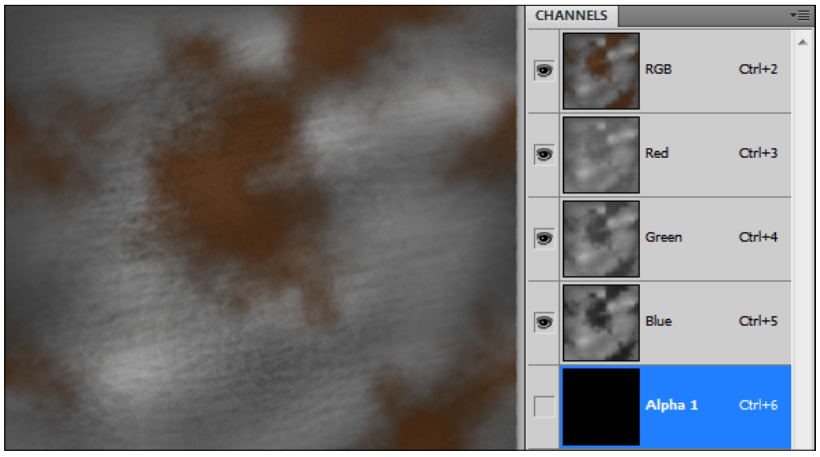
To create a specular material, follow these steps:

1. Create a new material called `rustyMetalMaterial`.
2. Select your material. In the **Inspector** view, under the material's name, use the drop-down menu to change its **Shader** value to **Specular**:

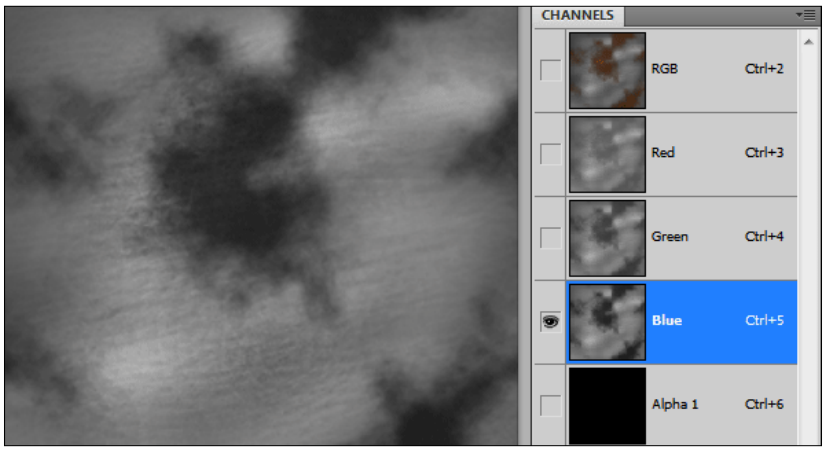


3. Open the base texture in your image editor (we'll use Adobe Photoshop to illustrate the next steps).
4. Open the **Channels** window (in Photoshop this can be done by navigating to **Window | Channels**).
5. There should be three channels: **Red**, **Green**, and **Blue**. Create a new channel. This will be the **Alpha** channel that controls the gloss level of our specular material:

www.it-ebooks.info



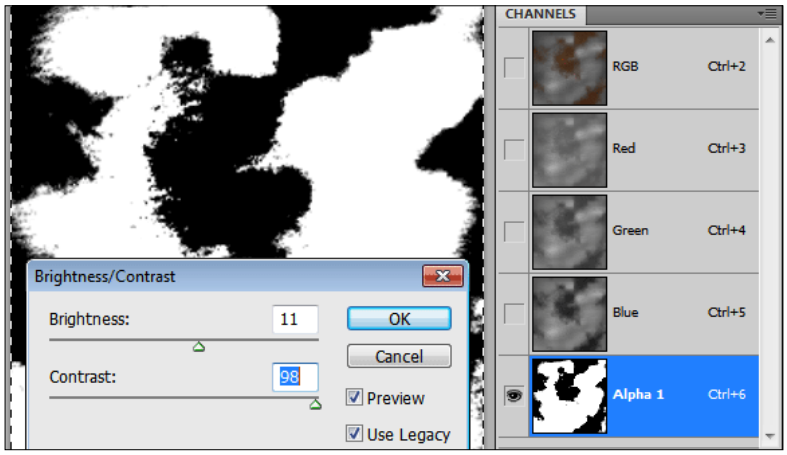
6. We want our **Alpha** channel to emphasize the contrast between rust and metal. Since **Blue** is the channel with higher level of contrast, we will select it on the **Channels** window and make all the other channels invisible. The resulting image will be displayed in grayscale, as shown in the following screenshot:



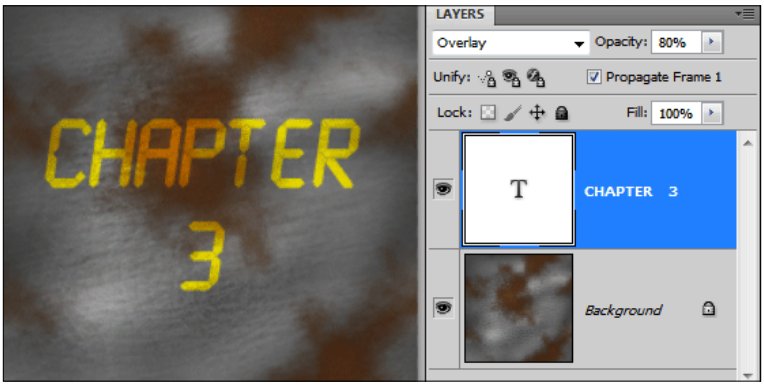
7. Select and copy all the image pixels (using *Ctrl + A*/*Ctrl + C* on Windows, or *Command + A*/*Command + C* on Mac OS).
8. In the **Channels** window, select the **Alpha** channel and paste the image you have copied.

www.it-ebooks.info

9. Now, intensify the contrast by going to **Image** | **Adjustments** | **Brightness/Contrast**:



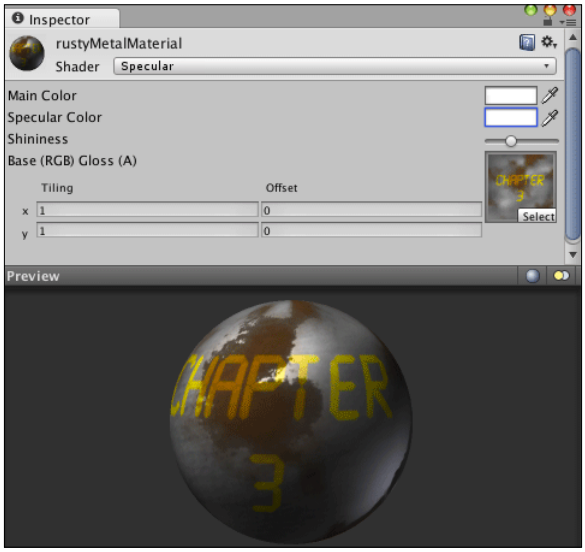
10. Still in the **Channels** window, bring the colors back by enabling all RGB channels.
11. In addition to the rusty metal plate, let's simulate a paint layer on the wall: create a type layer in yellow. You can type whatever you want.
12. Change the type layer's blending mode to **Overlay** and set its **Opacity** to **80%**:



13. Flatten your image (this is necessary, or Unity might discard the alpha channel you created and use the image layer transparent pixels as the alpha channel instead).
14. Save your work as a new image file (`.psd` or `.tga` formats are good options, as they keep the alpha channel).
15. Inside the Unity Editor, import your image file access through the **Assets** menu, clicking on **Import New Asset...**.

www.it-ebooks.info

16. In the **Project** view, choose the `rustyMetalMaterial` asset. Then, apply the texture you've made as a Base map (by either clicking on the **Select** button or dragging it from the **Project** view to the material slots). Your specular mapped material is ready:



How it works...

Unity is able to read four channels of a texture map: R (red), G (green), B (blue), and A (alpha). Specular shaders use RGB channels as the base texture (often called diffuse texture), while using the Alpha to set the specular brightness of the material according to each pixel's brightness level.

There's more...

The specular shader features some other parameters that are worth looking into: **Specular Color** and **Shininess**.

Changing the specular color of our material

If you ever get tired of that white shining spot, you can add some variety by changing the specular color.

Adjusting shininess to focus the highlight

Shining spots don't look the same on bowling balls (where they are highly concentrated) as they do on brushed metal surfaces (where they spread a bit more). This concentration level is controlled by the **Shininess** parameter.

www.it-ebooks.info

See also

- The *Creating transparency texture maps* recipe.

Creating transparency texture maps

If you want your player to partially see through an object, you'll need a transparent or semi-transparent material. Plastic film, cutouts, and grids are some of the artifacts you can produce with transparent texture maps.

Getting ready

For this recipe, we will create a material simulating a slide film. The main reason behind this choice is because this material will allows us to use full transparency to make rounded borders of the frame, and also use semi-transparency to illustrate the film itself. For the rest of this recipe, it is assumed you are capable of creating, using your image editor, an image like the one intended (basically, a beveled round-cornered rectangle containing a picture within). If not, please feel free to use the one included inside the `0423_03_04` folder.

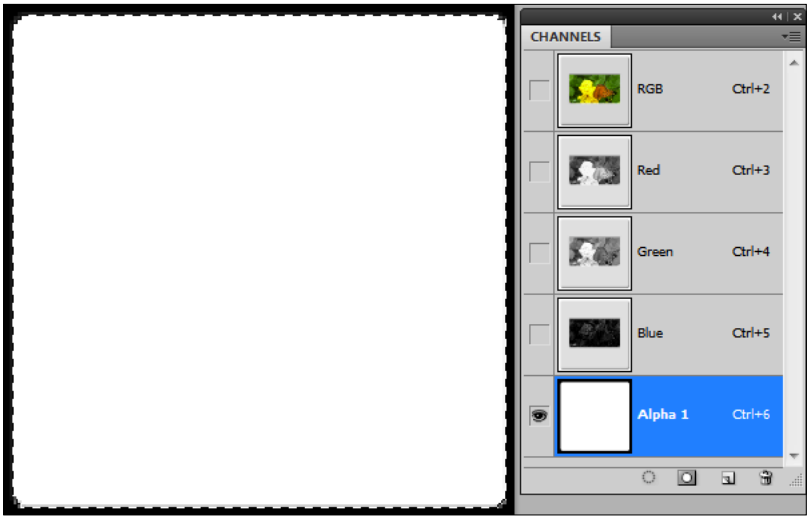
How to do it...

To create a transparent material, follow these steps:

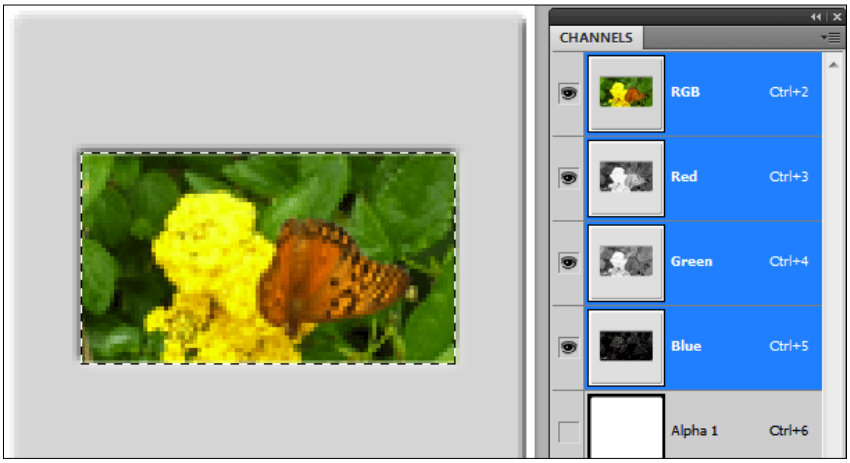
1. Create a new material named `slideMaterial`.
2. Select your material. In the **Inspector** view, under the material's name, use the drop-down menu to change its **Shader** value to **Transparent/Diffuse**.
3. Open the base texture in your image editor (we'll use Adobe Photoshop to illustrate the next steps).
4. Select the empty pixels around the rounded rectangle (this can be done with the **Magic Wand** tool with **Anti-alias** turned on and a **Tolerance** level of **0**).

www.it-ebooks.info

5. Open the **Channels** window (in Photoshop this can be done by going to **Window** | **Channels**).
6. There should be three channels: **Red**, **Green**, and **Blue**. Create a new channel. This will be the **Alpha** channel that controls the level of transparency, where darker levels are more transparent than brighter levels.
7. In the **Channels** window, select the **Alpha** channel and invert your selection (*Ctrl* + *Shift* + *I* for Windows and *Command* + *Shift* + *I* for Mac OS).
8. Using the **Paint Bucket** tool, paint the selection area white:

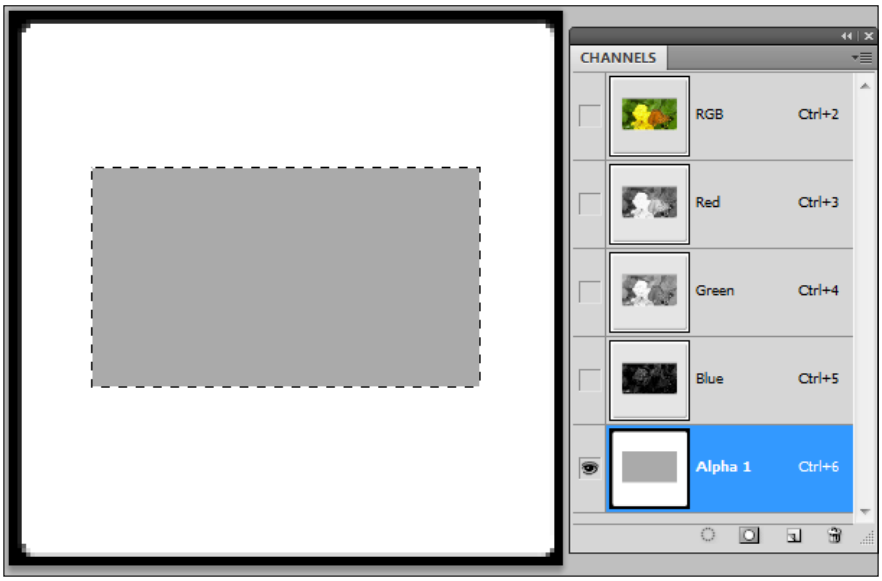


9. Reset your selection area (using *Ctrl* + *D*/*Command* + *D*). Now, create a rectangular selection around the actual photo:



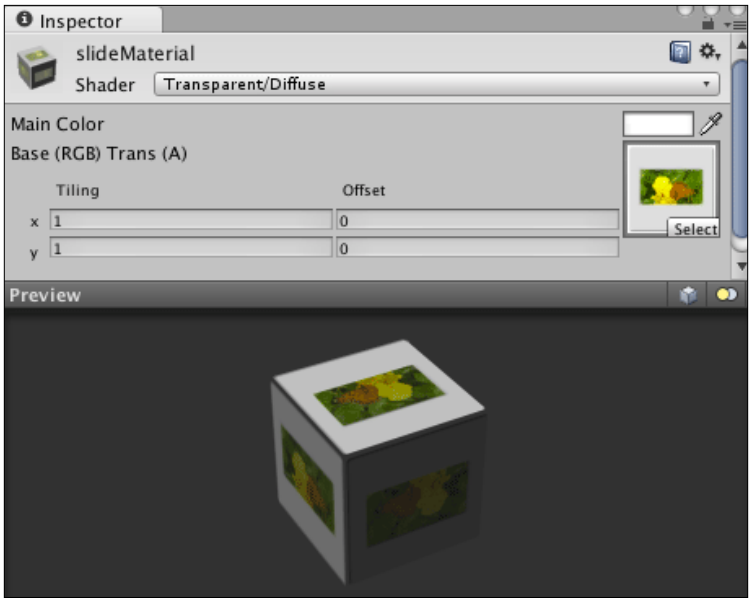
www.it-ebooks.info

10. Go back to the **Alpha** channel and fill the selected rectangle light gray (**R**, **G**, and **B** value around `170`, for instance):



11. Save your file (`.psd` or `.tga` formats are good options, as they keep the alpha channel).
12. Inside the Unity Editor, import your image file access by navigating to **Assets** | **Import New Asset...**.
13. In the **Project** view, choose **slideMaterial**. Then, select the image file you created as a Base map (by either clicking on the **Select** button or dragging it from the **Project** view to the material slots). Your transparent material is ready:

www.it-ebooks.info



How it works...

Unity is able to read four channels of a texture map: R (red), G (green), B (blue), and A (alpha). Transparent shaders use RGB channels as the base texture (often called diffuse texture), while using the Alpha to set the transparency of the material according to each pixel's brightness level. Transparent shaders in the **Cutout** subgroup will not render semi-transparency; they just allow texture pixels to be invisible or fully opaque.

There's more...

Unity has a range of transparent shaders that can be used to achieve different results.

Using Cutouts

There is a subgroup of transparent shaders named **Cutout**. If you don't need semi-transparency in your material, it might be a good idea to use a Cutout shader. They are faster and allow the object to cast and receive shadows.

www.it-ebooks.info

Applying Bumped Diffuse

To experiment, change the material's **Shader** value to **Transparent/Bumped Diffuse**. Then, apply the `slideNormalTexture.png` image file (included in the `0423_03_04` folder) as a normal bump map.

See also

- The *Creating a self-illuminated material* recipe.
- The *Creating specular texture maps* recipe.
- The *Solving transparent objects pop-ups* recipe.
- The *Disabling culling for a material* recipe.

Using cookie textures to simulate a cloudy outdoor

As it can be seen in many first-person shooters and survival horror games, lights and shadows can add a great deal of complexity to a scene, helping immensely to create the right atmosphere for the game. In this recipe, we will create a cloudy outdoor environment, lighting it using cookie textures.

Getting ready

If you don't have access to a image editor or prefer to skip the texture map elaboration in order to focus on the implementation, please use the `cloudCookie.tga` image file provided in the `0423_03_05` folder.

How to do it...

To simulate a cloudy outdoor environment, follow these steps:

1. In your image editor, create a new 512 x 512 pixel image.
2. Using black as foreground color and white as background color, apply the **Render Clouds** filter (in Photoshop, this is done by navigating to **Filter** | **Render** | **Clouds**):

www.it-ebooks.info

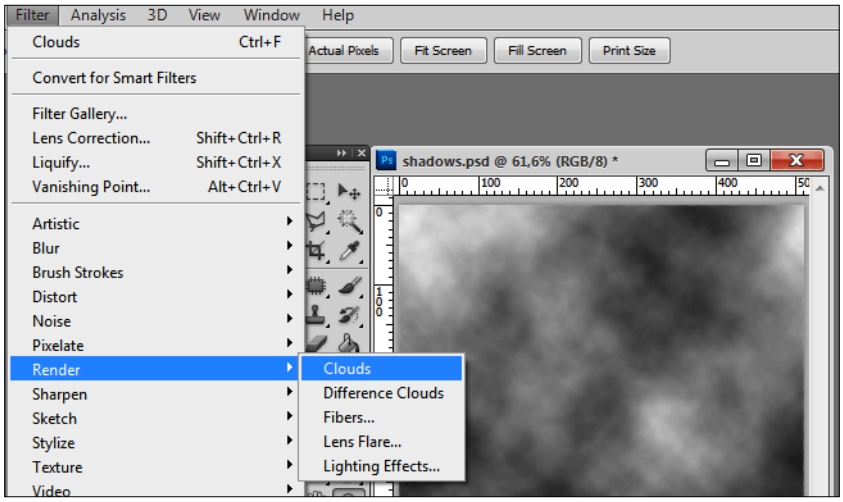
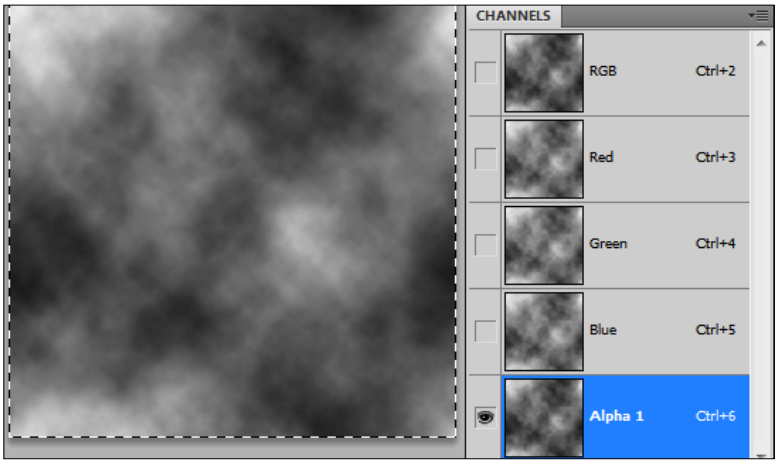


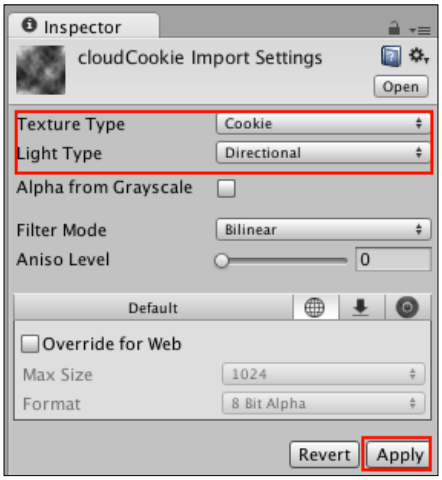
Image editors usually have a filter or command that renders clouds. If your image editor doesn't have such a capability, you can either paint it manually or use the `cloudCookie.tga` image file provided in the `0423_03_05` folder.

3. Select your entire image and copy it.
4. Open the **Channels** window (in Photoshop this can be done by navigating to **Window | Channels**).
5. There should be three channels: **Red**, **Green**, and **Blue**. Create a new channel. This will be the **Alpha** channel.
6. In the **Channels** window, select the **Alpha** channel and paste your image into it:



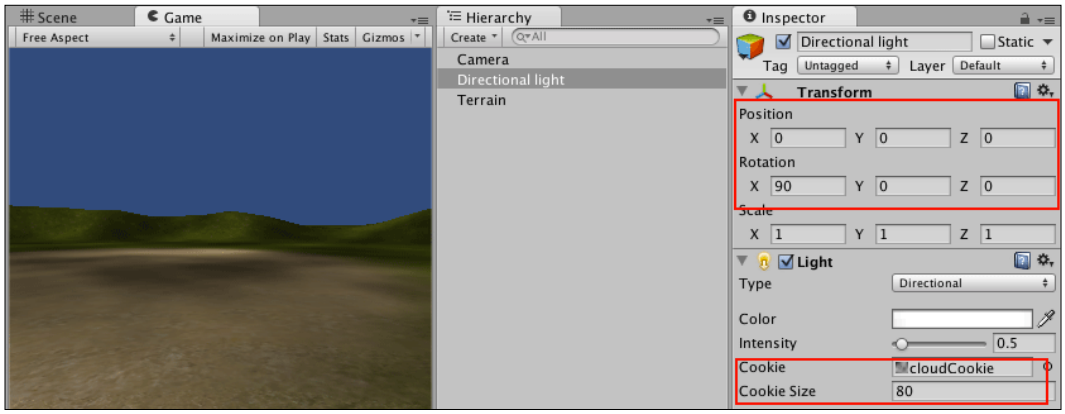
www.it-ebooks.info

7. Save your image file as `cloudCookie.PSD` or `cloudCookie.TGA`.
8. Import your image file into Unity and select it in the **Project** view.
9. In the **Inspector** view, set **Texture Type** to **Cookie**, and **Light Type** to **Directional** (as shown in the following screenshot):



10. We will need a surface to actually see the lighting effect. You can either add a plane to your scene (via the **GameObject** | **Create Other...** | **Plane** menu) or create a terrain (**Terrain** | **Create Terrain**) and edit it, if you wish.
11. Let's add a light to our scene. Since we want to simulate sunlight, the best option is to create a **Directional Light**. You can do that through the drop-down menu named **Create** in the **Hierarchy** window.
12. Using the **Transform** component of the **Inspector** view, reset the light's **Position** to `0`, `0`, `0` and its **Rotation** to `90`, `0`, `0`.
13. In the **Cookie** field, select the **cloudCookie** image that you imported earlier. Change the **Cookie Size** field to **80**, or a number that you feel as more appropriate to the scene's dimension. Please leave the **Shadow Type** as **No Shadows**:

www.it-ebooks.info



14. Now, we need a script to translate our light and, consequently, the cookie projection. Using the drop-down menu in the **Project** view, create a new **C# Script** and rename it to `MovingShadows`.

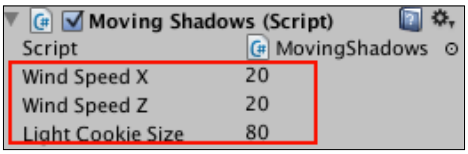
15. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class MovingShadows : MonoBehaviour
{
    public float windSpeedX;
    public float windSpeedZ;
    public float lightCookieSize; // make it equal to the light's
Cookie Size parameter in the inspector

    private Vector3 initPos;

    void Start()
    {
        initPos = transform.position;
    }

    void Update()
    {
        if (Mathf.Abs(transform.position.x) >= Mathf.
Abs(initPos.x) + lightCookieSize)
```


www.it-ebooks.info

```
            {
                Vector3 pos = transform.position;
                pos.x = initPos.x;
                transform.position = pos;
            }
            else
            {
                transform.Translate(Time.deltaTime * windSpeedX, 0, 0,
Space.World);
            }


            if (Mathf.Abs(transform.position.z) >= Mathf.
Abs(initPos.z) + lightCookieSize)
            {
                Vector3 pos = transform.position;
                pos.z = initPos.z;
                transform.position = pos;
            }
            else
            {
                transform.Translate(0, 0, Time.deltaTime * windSpeedZ,
Space.World);
            }
        }

    }
```

16. Save your script and apply it to the **Directional Light**.
17. Select the **Directional Light**. In the **Inspector** view, change the **Wind Speed X** and **Wind Speed Z** parameters to **20** (you can change these values as you wish). The **Light Cookie Size** parameter has to be changed to the exact amount of the light's cookie size (in our case, **80**):



18. Play your scene. The shadows should be moving.

www.it-ebooks.info

How it works...

With our script, we are telling the **Directional Light** to move across the x and z axis, causing the **Light Cookie** texture to be displaced as well. Also, we reset the light object to its original position whenever it travels a distance equal to or greater than the **Light Cookie Size** parameter. The light position must be reset to prevent it from travelling too far, causing problems in real time render and lighting. The **Light Cookie Size** parameter is used to ensure a smooth transition.

The reason we are not enabling shadows is because the light angle for the x axis must be 90 degrees (or there will be a noticeable gap when the light resets to the original position). If you want dynamic shadows in your scene, please add a second directional light.

There's more...

In this recipe we have applied a cookie texture to a directional light. But what if we were using Spot Light or Point Light cookies?

Creating Spot Light cookies

Unity documentation has an excellent tutorial on how to make Spot Light cookies. This is great to simulate shadows coming from projectors, windows, and so on. You can check it out at `http://unity3d.com/support/documentation/Manual/HOWTO-LightCookie.html`.

Creating Point Light cookies

If you want to use a cookie texture with a point light, you'll need to change the **Light Type** in the **Texture Importer** section of the **Inspector**.

Creating a color selection dialog

In-game customization and user-made content have been taking over games for a while. A very common feature is the ability of changing one's avatar color. In this recipe, we will create a dialog box that allows the player to change the object's color sliders that modify the red, green, and blue values of the material.

Getting ready

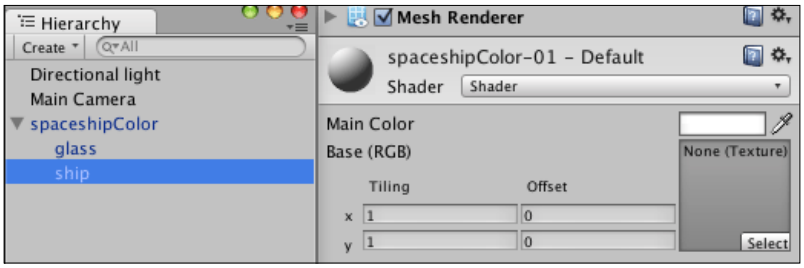
If you want to use our sample scene to follow this recipe, please import the package named `colorSelector`, available in the folder named `0423_03_06`.

www.it-ebooks.info

How to do it...

To create a color selection dialog, follow these steps:

1. Once the package has been imported, open the `colorSelection` scene.
2. Expand the **spaceshipColor** game object in the **Hierarchy** view and select its child named **ship**. This is the object that will have its material changed by our script:



3. In the **Project** view, create a new C# script. Rename it `ColorSelector` and open it in your editor.
4. Add the following code at the top of the script:

```
using UnityEngine;
using System.Collections;

public class ColorSelector : MonoBehaviour
{
    public float redValue = 1.0f;
    public float greenValue = 1.0f;
    public float blueValue = 1.0f;
    bool selectorOn = false;
    private float redReset = 1.0f;
    private float greenReset = 1.0f;
    private float blueReset = 1.0f;

    void OnMouseUp()
    {
        selectorOn = true;
    }

    void OnGUI()
    {
        if (selectorOn)
```

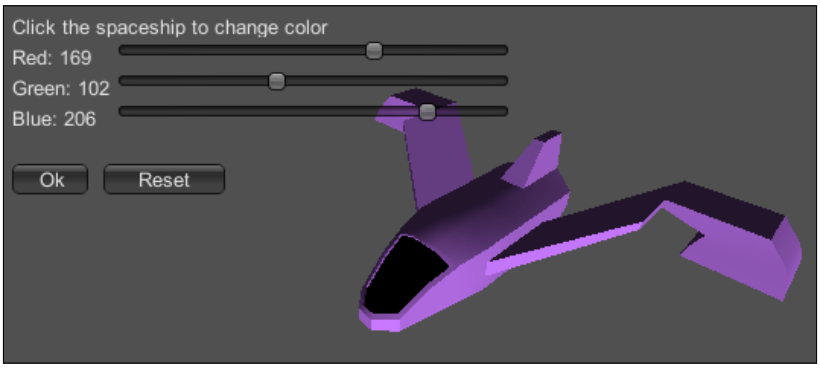

www.it-ebooks.info

```
        {
            GUI.Label(new Rect(10, 30, 90, 20), "Red: " + Mathf.
RoundToInt(redValue * 255));
            redValue = GUI.HorizontalSlider(new Rect(80, 30, 256,
20), redValue, 0.0f, 1.0f);
            GUI.Label(new Rect(10, 50, 90, 20), "Green: " + Mathf.
RoundToInt(greenValue * 255));
            greenValue = GUI.HorizontalSlider(new Rect(80, 50,
256, 20), greenValue, 0.0f, 1.0f);
            GUI.Label(new Rect(10, 70, 90, 20), "Blue: " + Mathf.
RoundToInt(blueValue * 255));
            blueValue = GUI.HorizontalSlider(new Rect(80, 70, 256,
20), blueValue, 0.0f, 1.0f);
            if (GUI.Button(new Rect(10, 110, 50, 20), "Ok"))
            {
                selectorOn = false;
                redReset = redValue;
                greenReset = greenValue;
                blueReset = blueValue;
            }
            if (GUI.Button(new Rect(70, 110, 80, 20), "Reset"))
            {
                redValue = redReset;
                greenValue = greenReset;
                blueValue = blueReset;
            }
            renderer.material.SetColor("_Color", new
Color(redValue, greenValue, blueValue, 1));
         }
        else
        {
            GUI.Label(new Rect(10, 10, 500, 20), "Click the
spaceship to change color");
        }
    }
}
```

5. Apply the `ColorSelector` script to the `ship` game object (and not its parent object, `spaceshipColor`).
6. Add a collider to the `ship` game object. Again, it is important that the collider is applied to `ship`, not its parent object. Adding a collider can be done by selecting the object and accessing **Component** | **Physics** | **Mesh Collider**.

www.it-ebooks.info

7. Play your scene and test your color selector:



How it works...

Besides assigning the sliders' values to the material color, we are keeping track of the last used color, storing its values into three variables. This is necessary in case the player wants to go back to the previous color after messing up with the sliders.

Also, we multiply those values by 255 in the text label so the player can read the RGB values in a more traditional way.

There's more...

With just a few twists, you could use this recipe to change other properties of your object's material (such as its transparency or emissive color, for instance).

Combining textures in real time through the GUI

The customization of avatars usually includes the selection of one or more textures that define its looks. In this recipe, we will implement a GUI that allows the player to create their avatar by combining two texture channels.

Getting ready

The 3D object and image files needed for this recipe are included in the `selectTexture` package, available inside the `0423_03_07` folder.

www.it-ebooks.info

How to do it...

To overlay textures, follow these steps:

1. Import the Unity package named `selectTexture` into your project.
2. Open the level named **selTextScene**.
3. Let's create our base material: in the **Project** view, use the drop down menu to create a new material. Name it `selectableMaterial`.
4. Change the **Shader** option of the **selectableMaterial** material to **Decal**. Then, apply the texture map named `face1` as the **Base** map and `prop1` as the **Decal** map, as shown in the following screenshot:

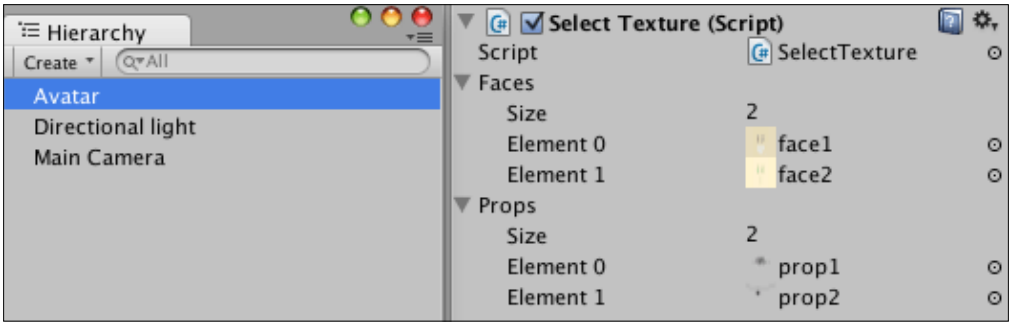
5. Apply the material to the `Avatar` game object. You can do this by dragging the material from the **Project** view into the game object's name in the **Hierarchy** view.
6. In the **Project** view, create a new C# script and rename it `SelectTexture`.
7. Open `SelectTexture` in your script editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class SelectTexture : MonoBehaviour
{
    public Texture2D[] faces;
    public Texture2D[] props;

    void OnGUI()
```


www.it-ebooks.info

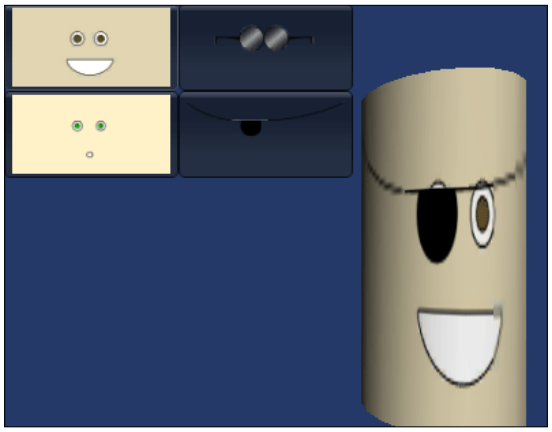
```
    {
        for (int i = 0; i < faces.Length; i++)
            if (GUI.Button(new Rect(0, i * 64, 128, 64),
faces[i]))
                ChangeMaterial("faces", i);

        for (int j = 0; j < props.Length; j++)
            if (GUI.Button(new Rect(128, j * 64, 128, 64),
props[j]))
                ChangeMaterial("props", j);
    }
    void ChangeMaterial(string category, int index)
    {
        if (category == "faces")
            renderer.material.mainTexture = faces[index];

        if (category == "props")
            renderer.material.SetTexture("_DecalTex",
props[index]);
    }
}
```

8. Save your script and apply it to `Avatar` game object.
9. In the **Inspector** view, change the **Size** value of both **Faces** and **Props** to `2`.
10. Set **Element 0** and **Element 1** of **Faces** to **face1** and **face2** respectively. Do the same for **Element 0** and **Element 1** of **Props** (changing it to **prop1** and **prop2** instead):



11. Play the scene. You will be able to select your texture combination by clicking on the appropriate buttons:

www.it-ebooks.info



How it works...

The script allows the user to create two collections of textures: one for the Base map (named **Faces**) and another one for the Decal map (named **Props**). When the scene is played, the textures are displayed inside GUI buttons which can be used to change the texture in the `Avatar` game object's material by calling the `ChangeMaterial` function. This function will receive, as parameters, both the category (either **Face** or **Prop**) and index of the image, assigning the correspondent texture map to the appropriate texture channel.

There's more...

This recipe could be easily adapted to change other parameters of different material shaders. Check Unity's online documentation to learn about other texture names at `http://unity3d.com/support/documentation/ScriptReference/Material.SetTexture.html`. Also, you might want to learn more about shaders by exploring the built-in shaders' source available at `http://unity3d.com/support/resources/assets/built-in-shaders`.

See also

- The *Creating a color selection dialog* recipe.

Highlighting materials at mouse over

Highlighting an object is a very effective way of letting players know they can interact with it. This is very useful in a number of game genres such as puzzles and point-and-click adventures, and it can also be applied to create 3D user interfaces.

www.it-ebooks.info

Getting ready

For this recipe, you'll need a 3D model and a 2D texture map. If you don't have them, please import the `highlight` package, available in the `0423_03_08` folder, into your project.

How to do it...

To highlight a material at mouse over, follow these steps:

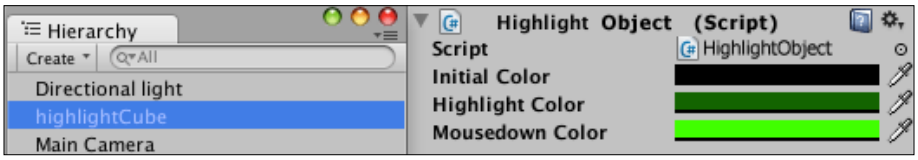
1. Import the Unity package and open the `highlightScene` scene.
2. In the **Hierarchy** view, select the 3D object to be highlighted (in our case, the object named `highlightCube`).
3. The **Inspector** view should display the game object's material. Using the drop-down menu, change its **Shader** option from **Diffuse** to **VertexLit**.
4. Apply the `baseBox` texture to the Base texture of the material.
5. Note that the **VertexLit** shader has a property called **Emissive Color**, which should be black as default. If you want a preview of what is going to happen later in the recipe, change that to green (but be sure to change it back to black).
6. We need to create a script. In the **Project** view, click on the **Create** drop-down menu and choose **C# Script**. Rename it `HighlightObject` and open it in your editor.
7. Replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class HighlightObject : MonoBehaviour
{
    public Color initialColor;
    public Color highlightColor;
    public Color mousedownColor;
    private bool mouseon = false;
    void OnMouseEnter()
    {
        mouseon = true;
        renderer.material.SetColor("_Emission", highlightColor);
    }
    void OnMouseExit()
    {
        mouseon = false;
        renderer.material.SetColor("_Emission", initialColor);
    }
```


www.it-ebooks.info

```
    void OnMouseDown()
    {
        renderer.material.SetColor("_Emission", mousedownColor);
    }
    void OnMouseUp()
    {
        if (mouseon)
            renderer.material.SetColor("_Emission",
highlightColor);
        else
            renderer.material.SetColor("_Emission", initialColor);
    }
}
```

8. Save your script and apply it to the `highlightCube` game object (in the **Hierarchy** view).
9. Select the **highlightCube** game object and, in the **Inspector** view, set **Highlight Color** to dark green (**R**: **0**, **G**: **100**, **B**: **0**) and **Mousedown Color** to light green (**R**:**0**, **G**: **255**, **B**: **0**):



10. Add a box collider to the **highlightCube** game object by navigating to **Component** | **Physics** | **Box Collider**.
11. Test the scene. The box should be highlighted when the mouse is over it (and even more when clicked).

How it works...

The box collider detects the mouse pointer over the object, working as a trigger for the emissive color value change. The `mouseon` Boolean variable is used to detect if the mouse button is released within or out of the box collider, changing its color accordingly.

There's more...

You can achieve other interesting results using other shaders, but be sure to address the changes to their particular material properties.

www.it-ebooks.info

Highlighting with self-illuminated shaders

Self-illuminated shaders will work if you replace `_Emission` with `_Color` in the script.

Using transparent shaders

Transparent shaders can be an interesting option. Remember you can change the transparency of the material by changing the Alpha value of its main color (which should be addressed as `_Color` in the script).

Animating textures by looping through array of materials (for example, simulated video)

There are occasions when you want the material of an object to be animated, for example a simulated computer screen in a spaceship control room, or a clickable object acting as a button to which you wish to draw the player's attention. While Unity Pro offers one solution of a video that can play as a texture, often this is overkill (or not an option if you are using Unity free). So another solution is to use code to change the texture of an object in real time.

Getting ready

You'll need a set of images that work as a sequence. It's usually a good idea to name them sequentially, for example, `computerScreen1.png`, `computerScreen2.png`, and so on.

How to do it...

To animate textures by looping through an array of materials, follow these steps:

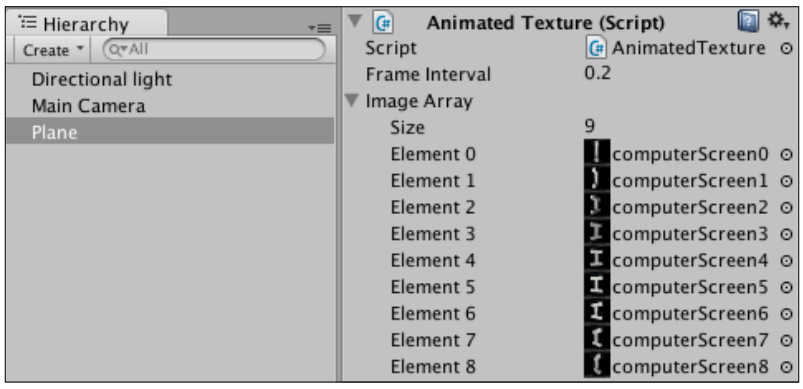
1. Start a new scene.
2. Create a **Plane** at (0,0,0), and set its rotation to (-90,0,0), so that it is squarely facing the camera.
3. Create a `ScriptClasses` project folder and in that folder create the `AnimatedTexture` C# script class (and paste the following code snippet inside this class):

```
// file: AnimatedTexture.cs
using UnityEngine;
using System.Collections;

public class AnimatedTexture : MonoBehaviour
{
  public float frameInterval = 0.9f;
  public Texture2D[] imageArray;
  private int imageIndex = 0;
```


www.it-ebooks.info

```
  private void Awake() {
    if (imageArray.Length < 1)
      Debug.LogError("no images in array!");
    else
      StartCoroutine( PlayAnimation() );
  }

  private IEnumerator PlayAnimation() {
    while( true ) {
      ChangeImage();
      yield return new WaitForSeconds(frameInterval);
    }
  }

  private void ChangeImage() {
    imageIndex++;
    imageIndex = imageIndex % imageArray.Length;
    Texture2D nextImage = imageArray[ imageIndex ];
    renderer.material.SetTexture("_MainTex", nextImage);
  }
}
```

4. Add the `AnimatedTexture` C# script class as a component of your **Plane** scene object.
5. Populate the public array with your sequence of images (you'll need to first set the `size` property of the array to the number of images).
6. To control the speed of the animation, in the **Inspector** view choose an appropriate value for the **Frame Interval** public variable:



www.it-ebooks.info

How it works...

The `imageArray` array stores the sequence of images. The `Awake()` method checks if there are some contents in the array, and starts the `PlayAnimation()` method as a co-routine.

The `PlayAnimation()` method runs an infinite loop, which calls the method to display the next image in the sequence, and then makes the `PlayAnimation()` method wait for the number of seconds set in `frameInterval` variable.

The `ChangeImage()` method needs to find the next image in the sequence and make the material of the parent game object use that image. The `imageIndex` instance variable always stores the array index of the currently displaying image. By adding 1 to this index value, and then applying a modulus operation, the result is the next highest array index, or zero, if the last image has already been displayed. A local `nextImage` variable is set to store the next image to be displayed from the array, and the main texture of the material of the parent game object is set to this image in the final statement of the method.

There's more...

You can also use a similar technique to change materials in real time.

Using materials rather than Texture2D images

If you prefer to work with materials rather than texture images, you could use the following code with an array of materials named `materialArray`, and change the shared material of the render of the parent game object, as illustrated in the final statement of this method:

```
private void ChangeMaterial()      {
  materialIndex++;
  materialIndex = (materialIndex % materialArray.Length);
  Material nextMaterial = materialArray[ materialIndex ];
  renderer.sharedMaterial = nextMaterial;
}
```

See also

- The *Highlighting materials at mouse over* recipe.

Disabling culling for a material

When creating a transparent or semi-transparent object, we might want to see its internal faces. However, by default, Unity transparent shaders make them invisible. In this recipe, we will edit one of Unity's built-in transparent shaders in order to make those faces visible to the user.

www.it-ebooks.info

Getting ready

In order to follow this recipe, you will need to download the source code to Unity's built-in shaders from `http://unity3d.com/unity/download/archive`. You will also need a texture with a transparency channel. We have provided one, named `grid.tga`, in the `0423_03_10` folder.

How to do it...

To disable culling for a material, follow these steps:

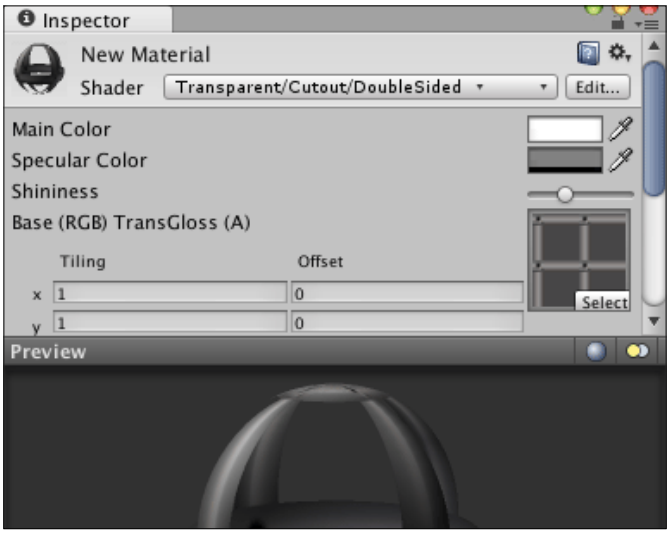
1. Open the archive containing the source code to Unity's built-in shaders and extract the file named `AlphaTest-BumpSpec.shader`, available within the `DefaultResources` folder, onto your desktop.
2. Rename the `AlphaTest-BumpSpec.shader` copy to `AlphaTest-DoubleSided.shader`. Then, open it in your code editor.
3. Change the first line to: `Shader "Transparent/Cutout/DoubleSided" {`.
4. Add the following line of code where indicated:

```
SubShader {

Tags {"Queue"="AlphaTest" "IgnoreProjector"="True" "RenderType"="T
ransparentCutout"}

  LOD 400
   // Line of code to be added:
  Cull Off
   // End of code to be added.
```

5. Save the file and import it into your Unity project.
6. Import the `grid.tga` texture.
7. Let's create a new material. Access the **Project** view, click on the **Create** drop-down menu and choose **Material**. Rename it to `Grid`.
8. Select the `Grid` material and, in the **Inspector** view, change its shader to `Transparent/Cutout/DoubleSided`.

www.it-ebooks.info

9. Apply the `grid.tga` texture as the **Base** texture for the `Grid` material. Your double-sided transparent material is ready for use. It can even cast shadows!



How it works...

The `Cull Off` command makes Unity render both front and back faces of the object. Although it works fine for our **Cutout** shader, using it with other types of transparent shaders might lead to unexpected results.

There's more...

Here are some additional resources and solutions:

Learning more about shaders

Unity's documentation includes several articles on shader programming. You can check them out at `http://docs.unity3d.com/Documentation/Components/SL-Reference.html`.

Solving the problem by editing the object's geometry

You can get similar results by duplicating the faces of your 3D models, and then flipping normals of the new geometry.

See also

- The *Creating transparency texture maps* recipe.

www.it-ebooks.info

4
Creating GUIs

In this chapter, we will cover:

- Displaying a digital clock
- Displaying an analogue clock
- Displaying a compass to show player direction
- Displaying a radar to indicate relative locations of objects
- Displaying images for corresponding integers
- Displaying images for corresponding floats and ranges
- Displaying a digital countdown timer
- Displaying a countdown timer graphically (5, 4, 3, 2, 1 – blast off)
- Displaying a countdown timer graphically as a pie-chart style clock
- Creating a message that fades away
- Displaying inventory texts for single object pickups
- Displaying inventory icons for single object pickups
- Managing inventories with a general purpose `PickUp` class
- Controlling the scrollbar with the mouse wheel
- Implementing custom mouse cursor icons

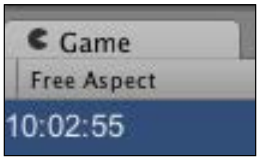
www.it-ebooks.info

Introduction

One element to the entertainment and enjoyment of most games is the quality of the visual elements, and an important part of this is the **Graphical User Interface** (**GUI**). GUI elements involve ways for the user to interact with the game (such as scroll wheels and cursors), and also ways for the game to present up-to-date information to the user (such as an inventory of what they are carrying, or the location of other objects in the game via a radar screen). This chapter is filled with GUI recipes to give you a range of examples and ideas for creating game GUIs.

Displaying a digital clock

Whether it is the real-world time, or an in-game clock, many games are enhanced by some form of clock or timer display. The most straightforward type of clock to display is a string composed of the integers for hours, minutes, and seconds:



How to do it...

To display a digital clock, please follow these steps:

1. Attach the following C# script class to the **Main Camera**:

```
// file: ClockDigital.cs
using UnityEngine;
using System.Collections;

using System;

public class ClockDigital : MonoBehaviour
{
  void OnGUI ()
  {
    DateTime time = DateTime.Now;
    string hour = LeadingZero( time.Hour );
    string minute = LeadingZero( time.Minute );
    string second = LeadingZero( time.Second );

    GUILayout.Label( hour + ":" + minute + ":" +  second);
```


www.it-ebooks.info

```
    }


    /**
     * given an integer, return a 2-character string
     * adding a leading zero if required
     */
    private string LeadingZero(int n)
    {
      return n.ToString().PadLeft(2, '0');
    }
  }
```

How it works...

Importing the `System` namespace allows the current time to be retrieved from the system using the `DateTime` class. The first line of the `OnGUI()` method retrieves the current time and stores it in a `time` `DateTime` object.

Integers for the hour, minute, and second are extracted from the `DateTime` object, and converted into two-character strings with the `LeadingZero()` method. This method pads out single digit values with a leading zero if required.

Finally, a string concatenating the hour, minute, and second strings with a colon separator is displayed via a GUI label.

There's more...

Some details you don't want to miss.

Converting to a 12-hour clock

If you prefer to display the hours in the 12-hour format, this can be calculated by finding the modulo 12 value of the 24-hour clock integer stored in the `DateTime` objects:

```
int hour12 = time.Hour % 12;
```

See also

- The *Displaying an analogue clock* recipe.

www.it-ebooks.info

Displaying an analogue clock

Analogue clocks are often more visually appealing, and often worth the little extra effort in creating the GUI elements for the clock-face and hands.

How to do it...

To display an analogue clock, please follow these steps:

1. Create a new scene, and add a **Directional light** in the direction of the camera.
2. Create the clock face, this is a **Cylinder** with the following properties:
   - **Position** (0, 0, 3)
   - **Rotation** (90, 0, 0)
   - **Scale** (13, 0.1, 13)
3. Create three differently colored or textured materials (for example, red, green, and blue), to be applied to the hands of the clock. Name these materials as `m_seconds`, `m_minutes`, and `m_hours`.
4. Create a new cube named `hand-seconds`, apply the `m_seconds` material, and then turn this into a long, thin second hand by setting the following properties:
   - **Position** (0, 2.5, 0)
   - **Scale** (0.1, 5, 1)
5. Create a new cube named `hand-minutes`, apply the `m_minutes` material, and turn this into a medium long, medium width minute hand by setting the following properties:
   - **Position** (0, 2, 1)
   - **Scale** (0.25, 4, 1)

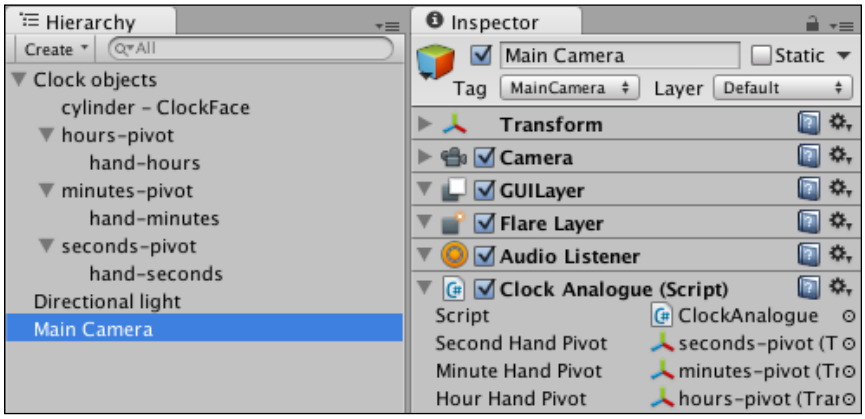
www.it-ebooks.info

6. Create a new cube named `hand-hours`, apply the `m_hours` material, and turn this into a short, wide hour hand by setting the following properties:
   - **Position** (0, 1.5, 2)
   - **Scale** (0.5, 3, 1)
7. You now need to create `pivot` game objects for each hand to rotate around. Create a new empty game object named `hours-pivot` at (0,0,0). Make this the parent of `hand-hours`. Create another named `minutes-pivot` at (0,0,0), and make this the parent of `hand-minutes`. Create a final object named `seconds-pivot` at (0,0,0), and make this the parent of `hand-seconds`.
8. Attach the following script class to the **Main Camera**:

```
// file: ClockAnalogue.cs
using UnityEngine;
using System.Collections;

using System;

public class ClockAnalogue : MonoBehaviour
{
  public Transform secondHandPivot;
  public Transform minuteHandPivot;
  public Transform hourHandPivot;

  private void Update()
  {
    DateTime time = DateTime.Now;
    float seconds = (float)time.Second;
    float minutes = (float)time.Minute;
    float hours12 = (float)time.Hour % 12;
    float angleSeconds = -360 * (seconds/60);
    float angleMinutes = -360 * (minutes/60);
    float angleHours = -360 * (hours12 / 12);

    // rotate each hand
    secondHandPivot.localRotation = Quaternion.Euler(0f, 0f,
angleSeconds);
    minuteHandPivot.localRotation = Quaternion.Euler(0f, 0f,
angleMinutes);
    hourHandPivot.localRotation = Quaternion.Euler(0f, 0f,
angleHours);
  }
}
```


www.it-ebooks.info

9. With the **Main Camera** selected in the **Hierarchy** view, drag the three parent "pivot" objects for each of the clock hands into the **Inspector** view, for the corresponding public **Transform** variables:



How it works...

The clock-face is constructed as follows:

- The clock-face is a very thin **Cylinder** object
- The three hands are stretched cubes.
- Each object is at a different z-axis position, so they appear to the viewer as follows:
  - The clock-face at the back
  - Then the fat hour hand
  - Then the middle sized minute hand
  - And closest to the viewer is the thin long second hand

Every frame's `Update()` method rotates all three hands about their clock-face center pivot game object by an angle proportionate to the seconds/minutes/hours of the `DateTime` object.

An angle in degrees is calculated as a proportion of 360 for each hand. This angle is used as the amount the object should be rotated about its z axis (calculated using the `Quaternion.Euler()` function provided by Unity).

Due to the LHS (left-hand-system) arrangement of the z axis in Unity, the rotation angles have to be negative—hence the negative 360 multiplied each time for the calculations for each clock hand angle in degrees.

www.it-ebooks.info

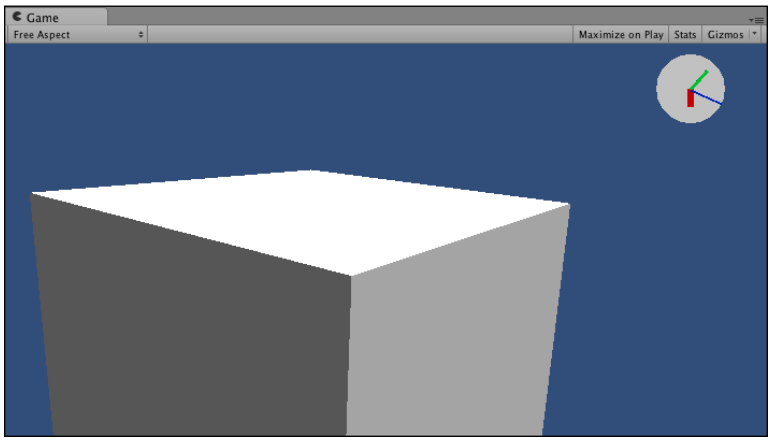
Each clock-hand has its original position arranged as the center of the clock-face, and then half the length of the hand added to the y-axis co-ordinate. For example, the clock face has a center of (0,0,0), and the second hand is a cube with a y-axis scale of 5, so its center is (0, 2.5, 0). This means that each hand, when rotated around the center of the clock-face, moves correctly. The width of the hands is determined by the x-axis scaling, so the second hand is thin (x-axis scale is 0.1) and long (y-axis scale is 5), while the hour hand is wide (x-axis scale is 0.5) and short (y-axis scale is 3). The wider short hands are further from the user (larger z-axis positions), and the thinner long hands are nearer the user, so that the wide hour hand can be seen behind the thinner minute and second hands even when pointing to the same clock-face position.

There's more...

Here are some details you don't want to miss.

Configuring a screen corner clock camera

Usually, a clock does not take up the whole screen, and one method of having a view of a `GameObject` that only takes up part of the screen, is by having a second camera. The following screenshot shows a scene where the Main Camera shows a cube, but a second camera (**Camera – clock**) has been created to display in the top-right corner of the screen:

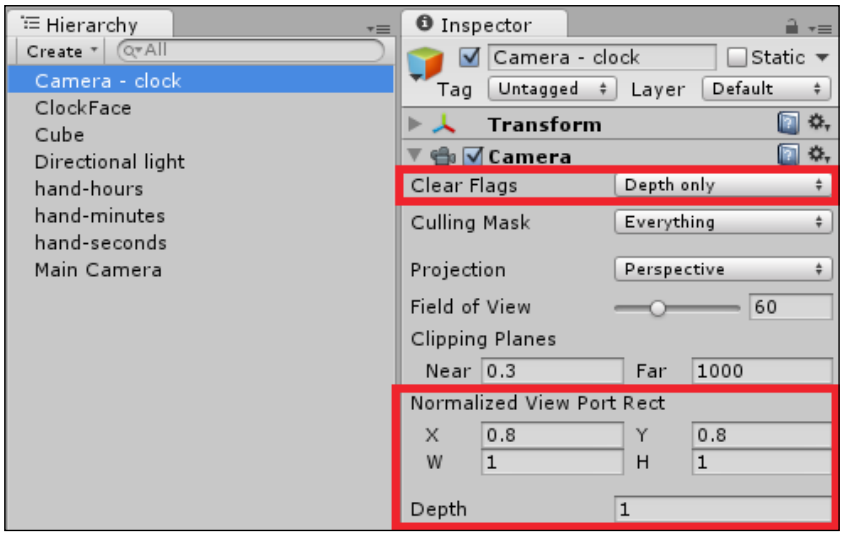


By default, cameras display to a "normalized viewport" of the whole screen (0,0)-(1,1). However, by setting the rectangle of a camera's normalized viewport to less than the whole screen, the desired effect can be created, as shown in the following screenshot. Our second **Camera – clock** camera has a viewport rectangle of (0.8,0.8)-(1,1) which defines the 20 percent top-right corner of the screen, as we can see in.

www.it-ebooks.info

Remove the **AudioListener** component from the second camera, since each scene should only have one such component.

To ensure the content of our clock is displayed in front of the **Main Camera** content, we need to set the Depth of our clock camera to 1, and to set its **Clear Flags** property to **Depth Only**.



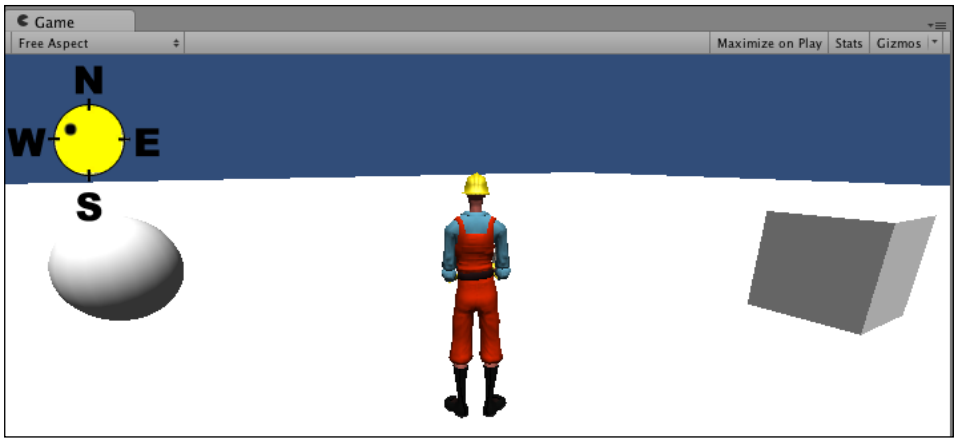
See also

- The *Displaying a digital clock* recipe.

Displaying a compass to show player direction

In games where players must refer to maps to confidently navigate large terrains, it can be very useful to display a compass in the GUI, presenting a real-time graphical indication of the direction the player is facing. The next screenshot shows a player's third-person controller looking North-West, in between the cube and sphere. The black "blip" circle on the compass indicates that the player is facing this compass direction.

www.it-ebooks.info



Getting ready

If you need a set of images for this recipe, you'll find two image files in the `0423_04_03` folder. One is a background image of a compass circle, and the other is a black circle image to represent the direction the player is facing.

How to do it...

To display a compass onscreen, please follow these steps:

1. Create a new scene, and add a directional light.
2. Create a new terrain, set **Size** to 2000 x 2000 and **Position** (-1000, 0, -1000).
3. Import the built-in **Character Controller** Unity package.
4. Add a **3rd Person Controller** to your scene at **Position** (0, 1, 0).
5. Create a cube in front (North) of the **3rd Person Controller** at (0, 1, 5).
6. Create a sphere to the left (West) of the **3rd Person Controller** at (-5, 1, 0).
7. Attach the following script class to the **Main Camera**:

```
// Compass.cs
using UnityEngine;
using System.Collections;

public class Compass : MonoBehaviour
{
 public Transform playerController;
 public Texture compassBackground;
```


www.it-ebooks.info

```
public Texture playerBlip;

private void OnGUI()
{
  // background displaying top left in square of 128 pixels
  Rect compassBackgroundRect = new Rect(0,0, 128, 128);
GUI.DrawTexture(compassBackgroundRect,compassBackground);
  GUI.DrawTexture(CalcPlayerBlipTextureRect(), playerBlip);
}

private Rect CalcPlayerBlipTextureRect()
{
  // subtract 90, so North (0 degrees) is UP
  float angleDegrees = playerController.eulerAngles.y - 90;
  float angleRadians = angleDegrees * Mathf.Deg2Rad;

  // calculate (x,y) position given angle
  // blip distance from center is 16 pixels
  float blipX = 16 * Mathf.Cos(angleRadians);
  float blipY = 16 * Mathf.Sin(angleRadians);

  // offset blip position relative to compass center (64,64)
  blipX += 64;
  blipY += 64;

  // stretch blip image to display in 10x10 pixel square
  return new Rect(blipX - 5, blipY - 5, 10, 10);
 }
}
```

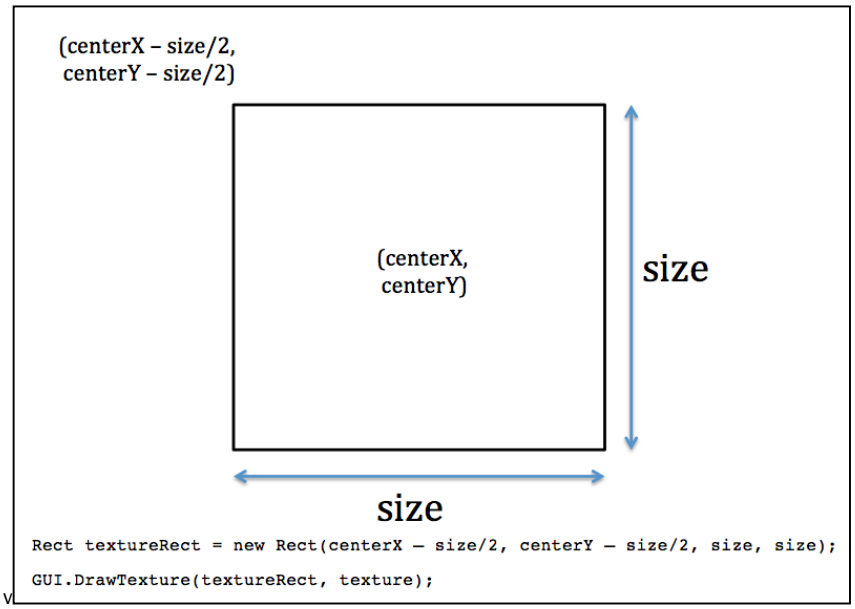
8. With the **Main Camera** selected in the **Hierarchy** view, drag the **3rd Person Controller**, the compass background image, and the image for the player's direction "blip" into the **Inspector** view for the three public variables.

How it works...

The `Compass` class needs three public variables, the first is a reference to the player's 3rd Person Controller, the other two are images for the compass background image (usually a circle of some kind with the compass letters displayed) and another image to indicate the direction the player is facing on the compass background.

www.it-ebooks.info

The `OnGUI()` method is called every frame, and at each frame the compass background needs to be displayed, followed by the image indicating the player's direction. The position of that "blip" image is calculated and returned by the `CalcPlayerBlipTextureRect()` method. Both the images are displayed using the `DrawTexture()` method of the `GUI` class, which expects a `Rect` object and the `texture` object. Many recipes result in us knowing the center co-ordinates of a square or rectangle and its size/width/height. Therefore (as the following figure shows) such calculations in this recipe are straightforward to calculate:



This recipe makes use of the "yaw" angle of rotation, which is rotation about the y-axis—that is, the direction a character controller is facing. This can be found in the "y" component of a GameObject's `eulerAngles` component of its transform. You can imagine looking from above, down at the character controller, and seeing what direction they are facing—this is just what we are trying to display graphically with the compass. In mathematics, an angle of zero indicates an east direction. To correct that, we need to subtract 90 degrees from the yaw Euler angle. The angle is then converted into radians, since that is required for the Unity trigonometry methods. We then multiply these `Sin()` and `Cos()` methods' results by the distance we want the blip to be displayed from the center of the compass circle (in our code example, this is `16` pixels). We add 64 pixels to these x and y co-ordinate results since that is the center of the compass' background image. These final values for `blipX` and `blipY` are the position on screen we wish the center of player's directional blip image to be displayed. Knowing the center of the blip image, and its width and height (both 10 pixels) allows the `Rect` object to be created for use by the `DrawTexture()` method called from `OnGUI()`.

www.it-ebooks.info

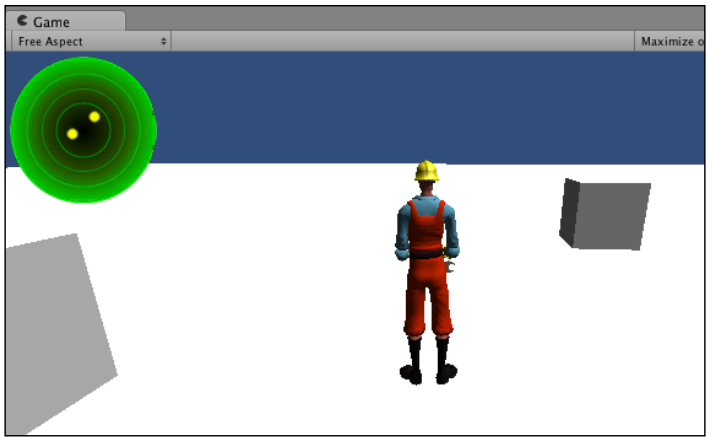
See also

- The *Displaying a radar to indicate relative locations of objects* recipe.

Displaying a radar to indicate relative locations of objects

Radars display the locations of other objects relative to the player, usually based on a circular display, where the center represents the player, and each graphical "blip' indicates, for each object, how far away and what relative direction it is to the player. Sophisticated radar displays will display different categories of object with different colored "blip" icons.

In the following screenshot, we can see two yellow circle "blips", indicating the relative position of the objects tagged `cube` near the player. The green circle radar background image gives the impression of an aircraft control tower radar.



Getting ready

If you need a set of images for this recipe, you'll find two image files in the `0423_04_04` folder. One is a background image of green radar circles, and the other is a yellow circle image to represent objects on the radar.

How to do it...

To display a radar showing relative object locations, please follow these steps:

1. Create a new scene, and add a directional light.
2. Create a new terrain, set **Size** to 2000 x 2000 and **Position** as (`-1000`, `0`, `-1000`).

www.it-ebooks.info

3. Import the built-in **Character Controller** Unity package.
4. Add a **3rd Person Controller** to your scene at position (0, 1, 0).
5. Create a cube in front of the **3rd Person Controller** at (0, 1, 5) and give this the `cube` tag.
6. Create a second cube to the left of the **3rd Person Controller** at (-5, 1, 0) and give this the `cube` tag.
7. Attach the following C# script class to the **Main Camera**:

```
// file: Radar.cs
using UnityEngine;
using System.Collections;

public class Radar : MonoBehaviour
{
 const float MAX_DISTANCE = 20f;
 const int RADAR_SIZE = 128;

 public Transform playerController;
 public Texture radarBackground;
 public Texture targetBlip;

 private void OnGUI()
 {
  // background displaying top left in square of 128 pixels
  Rect radarBackgroundRect = new Rect(0,0, RADAR_SIZE, RADAR_
SIZE);
  GUI.DrawTexture(radarBackgroundRect,radarBackground);

  // find all 'cube' tagged objects
  GameObject[] cubeGOArray = GameObject.
FindGameObjectsWithTag("cube");

  // draw blips for all within distance
  Vector3 playerPos = playerController.transform.position;
  foreach (GameObject cubeGO in cubeGOArray)
  {
    Vector3 targetPos = cubeGO.transform.position;
    float distanceToTarget = Vector3.
Distance(targetPos,playerPos);
    if( (distanceToTarget <= MAX_DISTANCE) )
      DrawBlip(playerPos, targetPos, distanceToTarget);
  }
 }
```


www.it-ebooks.info

```
 private void DrawBlip(Vector3 playerPos, Vector3 targetPos, float
distanceToTarget)
 {
  // distance from target to player
  float dx =  targetPos.x - playerPos.x;
  float dz =  targetPos.z - playerPos.z;

  // find angle from player to target
  float angleToTarget = Mathf.Atan2(dx,dz) * Mathf.Rad2Deg;

  // direction player facing
  float anglePlayer = playerController.eulerAngles.y;

  // subtract player angle, to get relative angle to object
  // subtract 90
  // (so 0 degrees (same direction as player) is UP)
  float angleRadarDegrees = angleToTarget – anglePlayer - 90;

  // calculate (x,y) position given angle and distance
  float normalisedDistance = distanceToTarget / MAX_DISTANCE;
  float angleRadians = angleRadarDegrees * Mathf.Deg2Rad;
  float blipX = normalisedDistance * Mathf.Cos(angleRadians);
  float blipY = normalisedDistance * Mathf.Sin(angleRadians);

  // scale blip position according to radar size
  blipX *= RADAR_SIZE/2;
  blipY *= RADAR_SIZE/2;

  // offset blip position relative to radar center
  blipX += RADAR_SIZE/2;
  blipY += RADAR_SIZE/2;

  // draw target texture at calculated location
  Rect blipRect = new Rect(blipX - 5, blipY - 5, 10, 10);
  GUI.DrawTexture(blipRect, targetBlip);
 }
}
```

8. With the **Main Camera** selected in the **Hierarchy** view, drag the **3rd Person Controller**, the radar background image, and the image for the cube's yellow "blip" into the **Inspector** view for the three public variables.

www.it-ebooks.info

How it works...

The following two constants are defined in our script code:

- `MAX_DISTANCE`: This specifies the maximum distance (in Unity "units") to which the objects are to be detected
- `RADAR_SIZE`: This specifies the size (in pixels) of the top-left square the radar will occupy

The `Radar` class needs three public variables:

- The first is a reference to the player's 3rd Person Controller
- The other two are images for the radar background image (usually a circle of some kind) and another image to indicate object locations on the radar background

Since there is a lot happening in the code for this recipe, each method will be described in its own section.

Displaying radar in the OnGUI() method

The `OnGUI()` method first displays the radar's background image. An array of `GameObjects` with the desired tag is retrieved, and iterated through. For each target `GameObject`, if its distance to the character controller is within `MAX_DISTANCE`, then the `DrawBlip()` method is called, with arguments of the `Vector3` positions of the player's character controller and the target game object.

Displaying radar object icons in the DrawBlip()method

The `DrawBlip()` method finds the x and z distances between the target and the player, and uses them to calculate the angle from the target to the player (using Unity's `Atan2()` inverse trigonometric function). The direction the player's character is facing is retrieved from its y-axis Euler angle (just as in the previous recipe). The player's direction angle is subtracted from the angle between the target and the player, since radar displays the relative angle from the direction the player is facing, to the target object. As usual, 90 degrees will need to be subtracted from the final angle, since we want zero degrees to be displayed as upwards in our GUI.

The `noramlisedDistance` value is calculated, which will always range between 0 and 1, by dividing the distance in pixels of the target from the player, by `MAX_DISTANCE`. The angle is converted into radians, since that is required for the Unity trigonometry methods. We then multiply these `Sin()` and `Cos()` results by the distance we want the blip to be displayed from the center of the radar circle, that is, `noramlisedDistance`. These coordinates are then converted to display in the top-left corner of the screen in a square of `RADAR_SIZE` pixels. Knowing the center of the blip image, and its width and height (both 10 pixels) allows the `Rect` object to be created for use by the `DrawTexture()` method called from the `GUI` class.

www.it-ebooks.info

There's more...

Here are some details you don't want to miss:

Different colored "blips" for different objects

Adding more public `Texture` variables allows for different icons to represent different categories of object on the radar. A `Texture` parameter can be added to the `DrawBlip()` method declaration, and that texture can be used in the `GUI.DrawTexture()` statement. This allows `DrawBlip()` to be called from different loops, each for objects tagged with a different string.

See also

- The *Displaying a compass to show player direction* recipe.

Displaying images for corresponding integers

While the first prototype of a user interface often simply displays game parameters as strings, engaging interfaces usually require customized graphical representations. A straightforward technique is to display an icon for each value of an integer, for example, a stickman, a heart, or ship for each life left, a bullet or magazine for ammo, and so on.

Getting ready

In the `0423_04_05` folder, you'll find an image of a heart, like the one shown here:

How to do it...

To display images instead of integer numbers, please follow these steps:

1. Attach the following script class to the **Main Camera**:

```
// file: PlayerGUI.cs
using UnityEngine;
using System.Collections;
```


www.it-ebooks.info

```
public class PlayerGUI : MonoBehaviour
{
  public Texture heartImage;
  private int livesLeft = 5;

  private void OnGUI()
  {
    Rect r = new Rect(0,0,Screen.width, Screen.height);
    GUILayout.BeginArea(r);
    GUILayout.BeginHorizontal();

    ImagesForInteger(livesLeft, heartImage);

    GUILayout.FlexibleSpace();
    GUILayout.EndHorizontal();
    GUILayout.EndArea();
  }

  private void ImagesForInteger(int total, Texture icon)
  {
    for(int i=0; i < total; i++)
    {
      GUILayout.Label(icon);
    }
  }
}
```

2. With the **Main Camera** selected in the **Hierarchy** view, drag the heart image into the **Inspector** view for the public variable.

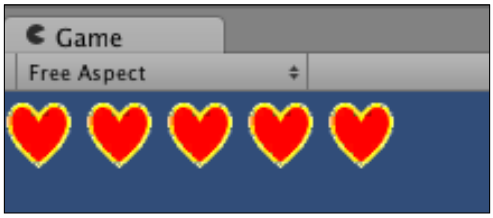
How it works...

A public `Texture` variable will store a heart image. A private `livesLeft` integer variable stores the game parameter to be graphically represented.

The `OnGUI()` method sets up a basic `GUILayout` area that will display contents horizontally, and left aligned (via an `Area`, `Begin-End Horizontal`, and a `FlexibleSpace` after the contents are displayed). A call is made to the `ImagesForInteger()` method, passing in the `livesLeft` integer and the heart `Texture`.

www.it-ebooks.info

The `ImagesForInteger()` method is a straightforward `for` loop, which repeatedly displays the provided image argument via `GUILayout.Label()`. The image is displayed as many times as defined by the integer argument total.



There's more...

Here are some details you don't want to miss:

Choosing between GUI layout alternatives: GUILayout versus GUI

You may prefer to use `GUI` rather than `GUILayout`—this saves having to set up an `Area`, `Horizontal`, and `FlexibleSpace` calls. However, it does require a horizontal pixel position to be maintained as part of the loop, making the `ImagesForInteger()` method a little more complicated.

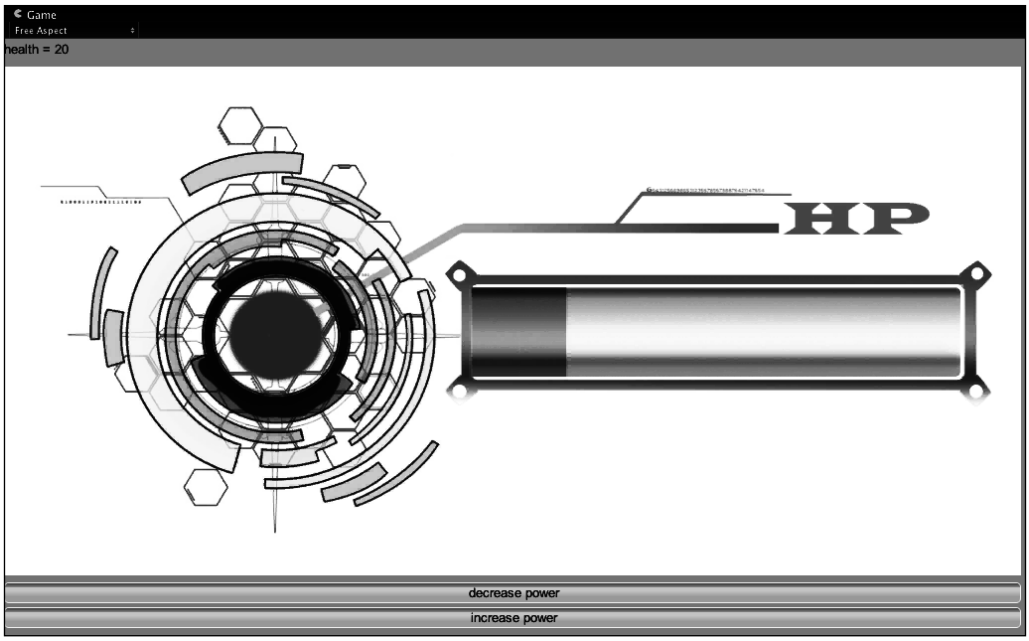
See also

- The *Displaying images for corresponding floats and ranges* recipe.
- The *Displaying a countdown timer graphically (5, 4, 3, 2, 1 – blast off)* recipe.
- The *Displaying a countdown timer graphically as a pie-chart style clock* recipe.

Displaying images for corresponding floats and ranges

As the previous recipe illustrates, displaying images for integers with one-to-one correspondence is straightforward. However, sometimes there may be a small number of images to help highlight the "zone" of a numeric variable. Examples might include health points—with green (very healthy) and red (little health remaining) displays. Whether values are represented as floats (for example, 0.0 – 1.0) or ranges within larger numbers (for example, 0 – 50), a general solution is to identify the sub-range (zone) of the numeric value and display the appropriate corresponding image.

www.it-ebooks.info



Getting ready

In the `0423_04_06` folder, you'll find a series of PNG images.

How to do it...

To display images corresponding to floats and ranges, please follow these steps:

1. Attach the following script class to the **Main Camera**:

```
// file: HealthBar.cs
using UnityEngine;
using System.Collections;

public class HealthBar : MonoBehaviour
{
  const int MAX_HEALTH = 100;
  public Texture2D bar00;
  public Texture2D bar10;
  public Texture2D bar20;
  public Texture2D bar30;
  public Texture2D bar40;
  public Texture2D bar50;
```


www.it-ebooks.info

```
  public Texture2D bar60;
  public Texture2D bar70;
  public Texture2D bar80;
  public Texture2D bar90;
  public Texture2D bar100;

  private int healthPoints = MAX_HEALTH;

private void OnGUI()
{
      GUILayout.Label("health = " + healthPoints);
    float normalisedHealth = (float)healthPoints / MAX_HEALTH;
      GUILayout.Label( HealthBarImage(normalisedHealth) );

      bool decButtonClicked = GUILayout.Button("decrease power");
      bool incButtonClicked = GUILayout.Button("increase power");

    if( decButtonClicked )
      healthPoints -= 5;

      if( incButtonClicked )
      healthPoints += 5;
    }

  private Texture2D HealthBarImage(float health)
   {
    if( health > 0.9 ){  return bar100;  }
    else if( health > 0.8 ){  return bar90;  }
    else if( health > 0.7 ){  return bar80;  }
    else if( health > 0.6 ){  return bar70;  }
    else if( health > 0.5 ){  return bar60;  }
    else if( health > 0.4 ){  return bar50;  }
    else if( health > 0.3 ){  return bar40;  }
    else if( health > 0.2 ){  return bar30;  }
    else if( health > 0.1 ){  return bar20;  }
    else if( health > 0 ){  return bar10;  }
    else{  return bar00;  }
  }
}
```

2. With the **Main Camera** selected in the **Hierarchy** view, drag each of the images into the **Inspector** view for the corresponding public variable.

www.it-ebooks.info

How it works...

The private `healthPoints` integer variable stores the game parameter to be graphically represented. The `MAX_HEALTH` constant defines the value of `healthPoints`, which indicates full health. The 11 public image variables store different versions of the health bar image to correspond to different health ranges. For example, `bar00` corresponds to zero health remaining, so it appears as an image with warning red coloring.

The core function in this recipe is a method being called that returns a `Texture` image object corresponding to the current value of `healthPoints`. A normalized value is created (which will always range between 0.0 and 1.0) by dividing `healthPoints` by `MAX_HEALTH`. This normalized float is passed to the `HealthBarImage()` method, which returns an image; the image is displayed via a `Label()` method call.

The `HealthBarImage()` method uses a sequence of `if` statements to decide which image to return. The provided health value is tested in a sequence of decreasing boundary values—`0.9` tested first, then `0.8`, and so on. Ranges are implied by these values, so if the normalized health value is from `0.9` to `1.0`, the image in the `bar100` variable will be displayed.

By normalizing values into the range 0.0 – 1.0, methods can be re-used in many different circumstances. So whether the health points for one game are from 0 – 50, or 0 – 10, or 0 – 500, the same code can be used, by always normalizing values by dividing them by their maximum value to get the 0.0 – 1.0 range.

If ranges corresponding to images are always of the same size, then an even more general purpose method could be created that takes as input a normalized value, and an array of images, and determines the ranges depending on the size of the array.

There's more...

Here are some details you don't want to miss:

Cleaning up code with an array of textures

While having lots of public `Texture2D` variables is straightforward, cleaner code can be achieved by a single public array of `Texture2D` objects (in the **Inspector** view, you'll need to set the size of the array to `11` for this example):

```
// public array
public Texture2D[] barImageArray;
```


www.it-ebooks.info

The `if` statements inside the `HealthBarImage()` method now return values from this array, for example:

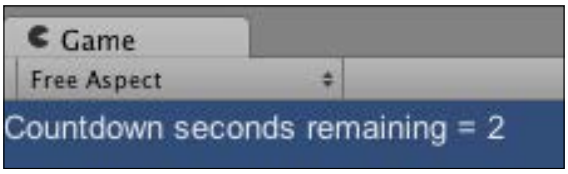
```
if( health > 0.9 ){  return barImageArray[10];  }
```

See also

- The *Image display for corresponding integers* recipe.
- The *Displaying a countdown timer graphically (5, 4, 3, 2, 1 – blast off)* recipe.
- The *Displaying a countdown timer graphically as a pie-chart style clock* recipe.

Displaying a digital countdown timer

Many games involve players completing tasks or getting bonus points within a set time. This recipe shows how to create a basic countdown timer, similar to the one shown in the following screenshot:



How to do it...

To display a digital countdown timer, please follow this step:

1. Attach the following C# script class to the **Main Camera**:

```
// file: CountdownTimer.cs
using UnityEngine;
using System.Collections;

public class CountdownTimer : MonoBehaviour {
  private float secondsLeft = 3f;

  private void OnGUI(){
    if( secondsLeft > 0)
      GUILayout.Label("Countdown seconds remaining = " + (int)
secondsLeft   );
    else
      GUILayout.Label("countdown has finished");
```


www.it-ebooks.info

```
    }

    private void Update(){
      secondsLeft -= Time.deltaTime;
    }
}
```

How it works...

Each `secondsLeft` frame is decremented by the time since the last frame (`Time.deltaTime`). When this variable goes below zero, the timer has finished. The value in the variable can be presented to the user as an integer through a simple cast statement, that is, `(int) secondsLeft`.

See also

- The *Displaying a countdown timer graphically (5, 4, 3, 2, 1 – blast off)* recipe.

Displaying a countdown timer graphically (5, 4, 3, 2, 1 – blast off)

Timers can have more effects when displayed graphically. This recipe illustrates a simple way to display a rocket ship-style countdown, using images for each second (5, 4, 3, 2, 1) and a final image for the "blast off". The following screenshot shows an example of this:

www.it-ebooks.info

Getting ready

If you need a set of images for this recipe; you'll find a series of PNG images in the `0423_04_08` folder.

How to do it...

To display a countdown timer graphically, please follow these steps:

1. Attach the following script class to the **Main Camera**:

```
// file: CountdownGraphical.cs
using UnityEngine;
using System.Collections;

public class CountdownGraphical : MonoBehaviour {
  public Texture2D imageDigit1;
  public Texture2D imageDigit2;
  public Texture2D imageDigit3;
  public Texture2D imageDigit4;
  public Texture2D imageDigit5;
  public Texture2D imageBlastOffText;

  private int countdownTimerDelay;
  private float countdownTimerStartTime;

  void Awake(){
    CountdownTimerReset( 5 );
  }

  void OnGUI(){
    GUILayout.Label( CountdownTimerImage() );
  }

  void CountdownTimerReset(int delayInSeconds){
    countdownTimerDelay = delayInSeconds;
```


www.it-ebooks.info

```
    countdownTimerStartTime = Time.time;
  }

  int CountdownTimerSecondsRemaining(){
    int elapsedSeconds = (int)(Time.time -
countdownTimerStartTime);
    int secondsLeft = (countdownTimerDelay - elapsedSeconds);
    return secondsLeft;
  }

  Texture2D CountdownTimerImage(){
    switch( CountdownTimerSecondsRemaining() ){
    case 5:
      return imageDigit5;
    case 4:
      return imageDigit4;
    case 3:
      return imageDigit3;
    case 2:
      return imageDigit2;
    case 1:
      return imageDigit1;
    default:
      return imageBlastOffText;
    }
  }
}
```

2. With the **Main Camera** selected in the **Hierarchy** view, drag each of the images into the **Inspector** view for the corresponding public variable.

How it works...

An alternative method of implementing a countdown timer algorithm is presented in this recipe (whereby the countdown progress is measured by comparing the current elapsed seconds (`Time.time`) against the time when the timer was started—`countdownTimerStartTime`).

In this recipe, the `OnGUI()` method displays a label with the image returned from the `CountdownTimerImage()` method. This method uses a `switch` statement to return the corresponding image to the current number of seconds remaining. Six images are public variables, and so can be set via the **Inspector**.

www.it-ebooks.info

See also

- The *Displaying a digital countdown timer* recipe.
- The *Displaying images for corresponding floats and ranges* recipe.

Displaying a countdown timer graphically as a pie-chart style clock

Another way to display a timer graphically is as a kind of clock, so the player can see the time running out. The following example shows elapsed time as the red portion of the circle:

Getting ready

In the `0423_04_09` folder, you'll find a series of pie-chart PNG images.

How to do it...

To display a pie-chart style graphical countdown timer, please follow these steps:

1. Attach the following script class to the **Main Camera**:

```
// file: CoundownClock.cs
using UnityEngine;
using System.Collections;

public class CoundownClock : MonoBehaviour
{
  public int timerTotalSeconds = 5;

  public Texture2D time0;
  public Texture2D time10;
  public Texture2D time20;
```


www.it-ebooks.info

```
  public Texture2D time30;
  public Texture2D time40;
  public Texture2D time50;
  public Texture2D time60;
  public Texture2D time70;
  public Texture2D time80;
  public Texture2D time90;
  public Texture2D time100;

  private int countdownTimerDelay;
  private float countdownTimerStartTime;

  private void Awake(){
    CountdownTimerReset( timerTotalSeconds );
  }

  private void CountdownTimerReset(int delayInSeconds){
    countdownTimerDelay = delayInSeconds;
    countdownTimerStartTime = Time.time;
  }

  private void OnGUI(){
      float propertionRemaining = (CountdownSecondsLeftFloat() /
timerTotalSeconds);
      GUILayout.Label( TimeRemainingImage(propertionRemaining) );
    }

  private float CountdownSecondsLeftFloat(){
    float elapsedSeconds = Time.time - countdownTimerStartTime;
    float secondsLeft = countdownTimerDelay - elapsedSeconds;
    return secondsLeft;
  }

  private Texture2D TimeRemainingImage(float propertionRemaining)
  {
    if( propertionRemaining > 0.9 ){  return time100;  }
    else if( propertionRemaining > 0.8 ){  return time90;  }
    else if( propertionRemaining > 0.7 ){  return time80;  }
    else if( propertionRemaining > 0.6 ){  return time70;  }
    else if( propertionRemaining > 0.5 ){  return time60;  }
    else if( propertionRemaining > 0.4 ){  return time50;  }
    else if( propertionRemaining > 0.3 ){  return time40;  }
    else if( propertionRemaining > 0.2 ){  return time30;  }
```


www.it-ebooks.info

```
        else if( propertionRemaining > 0.1 ){  return time20;  }
        else if( propertionRemaining > 0 ){  return time10;  }
        else{     return time0;  }
    }
}
```

2. With the **Main Camera** selected in the **Hierarchy** view, drag each of the images into the **Inspector** view for the corresponding public variable.

How it works...

This recipe is a combination of the basic countdown timer and the display of images for floats and ranges. Key aspects of this recipe include:

- The total number of seconds needs to be known by the `OnGUI()` method, so that a normalized proportion of time remaining can be calculated (in the range 0.0 – 1.0).
- The seconds left needs to be returned as a float, so the clock will work fine for short time intervals or long ones, and so the `CountdownSecondsLeftFloat()` method returns the remaining seconds as a float rather than an integer value, as in the previous recipes.
- The `TimeRemainingImage()` method takes in a float argument in the range 0.0 – 1.0, and uses `if` statements to return the image corresponding to the proportion of the total seconds remaining in the countdown.

There's more...

Here are some details you don't want to miss:

Creating your own images with a spreadsheet

The pie-chart clock style images were created using a spreadsheet, and by just changing the two values. Stacked bar charts are another useful spreadsheet graph for timer or health graphical images.

See also

- The *Displaying a digital countdown timer* recipe.
- The *Displaying a countdown timer graphically (5, 4, 3, 2, 1 – blast off)* recipe.
- The *Displaying images for corresponding floats and ranges* recipe.

www.it-ebooks.info

Creating a message that fades away

Often, we want to show the user a message briefly, and then have that message disappear. A nice way to do this is to have the message fade away, and then destroy itself when is it no longer visible. The following screenshot is an example of such a message:

How to do it...

To display a message that fades away, please follow these steps:

1. Create a new `GUIText` object, position it and change the font size and text as desired.
2. Attach the following script class to this `GUIText` object:

```
// file: FadingMessage
using UnityEngine;
using System.Collections;
public class FadingMessage : MonoBehaviour {
  const float DURATION = 2.5f;

  private void Update() {
    if( Time.time > DURATION){
      Destroy(gameObject);
    }

    Color newColor = guiText.material.color;

    float proportion = (Time.time / DURATION);
    newColor.a = Mathf.Lerp(1, 0, proportion);
    guiText.material.color = newColor;
  }
}
```


www.it-ebooks.info

How it works...

The `DURATION` constant defines the number of seconds the image is to be visible for. Once the scene has been running for this many seconds, the parent game object is destroyed. Dividing the elapsed time by `DURATION` gives the proportion of the life for this fading message (between 0.0 and 1.0). The `Mathf.Lerp()` method takes such a proportion as its third argument, and gives the corresponding value scaled between the first and second arguments—in this case, we wish to lerp from 1 (full alpha visibility) to 0 (invisible). This alpha value is assigned to a copy of the `guiText` material's color, and then this new color object is reassigned to the `guiText` object's material.

There's more...

Here are some details you don't want to miss:

Facilitating several messages by using a prefab

By creating a prefab, and placing a copy of the `GUIText` object with this script attached, messages can be easily created as required by the main game manager. Assuming that the `fadingMessagePrefab` public variable has been assigned a value of the prefab via the Inspector view, then the following method could be called whenever a fading message is required. This will then create an instance of the `GUIText` prefab and sets its text property to whatever message is passed in to the method:

```
    private void CreateMessage(string message)
    {
      GameObject fadingMessageGO = (GameObject)Instantiate(fadingMessag
ePrefab);

      fadingMessageGO.guiText.text = message;
    }
```

Improving efficiency by re-using a single GUIText message

Another strategy is to have a single `GUIText` object always present in the scene, but not always visible. Each time a message is to be displayed to the user, the new text is set and the alpha is set back to 1 (or its desired starting value), and then it will start fading again.

Displaying inventory texts for single object pickups

Many games involve the player picking up items. Perhaps the most common is the need to pick up a key to be able to pass through a doorway, as shown in the following screenshot:

www.it-ebooks.info

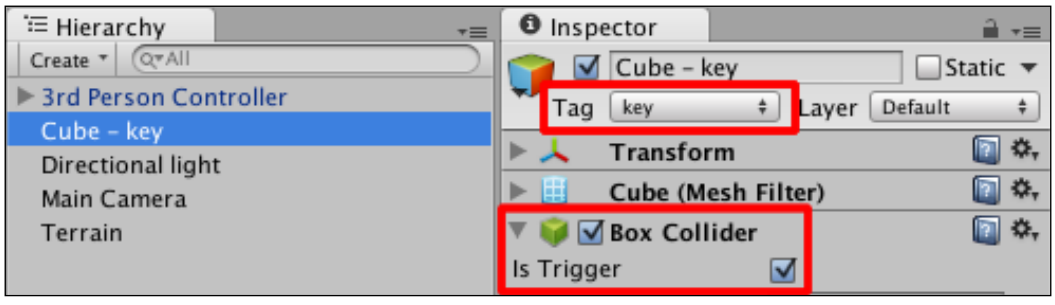
Getting ready

In the `0423_04_11` folder you'll find some key images, and a crisscross image to apply to the terrain.

How to do it...

To display a text description of inventory objects, please follow these steps:

1. Create a new scene, and add a directional light.
2. Create a new terrain with **Size** set to 2000 x 2000 and **Position** (-1000, 0, -1000). Now, apply a texture to this terrain.
3. Create a cube named **Cube – key** at position (0, 1, 5) with scale (2, 2, 2).
4. Tag **Cube–key** with the string **key**, and tick its **Is Trigger** checkbox, as shown in the following screenshot:



5. Add a key image to **Cube–key**.

www.it-ebooks.info

6. Import the built-in **Character Controller** Unity package and add a **3rd Person Controller** to your scene at position (0, 1, 0).
7. Attach the following script class to your character controller:

```
// file: PlayerInventory
using UnityEngine;
using System.Collections;

public class PlayerInventory : MonoBehaviour
{
  private bool isCarryingKey = false;

  private void OnGUI()
  {
    string keyMessage = "(bag is empty)";
    if( isCarryingKey )
    {
      keyMessage = "carrying: [ key ]";
    }

    GUILayout.Label ( keyMessage );
  }

  private void OnTriggerEnter(Collider hitCollider)
  {
    if( "key" == hitCollider.tag )
   {
      isCarryingKey = true;
      Destroy ( hitCollider.gameObject );
    }
  }
}
```

How it works...

The `isCarryingKey` Boolean variable represents whether or not the player is carrying the key at any point in time.

In the `OnGUI()` method the contents of the `keyMessage` string is displayed via the `GUILayout.Label()` method. The default value of this string tells the user that the players bag is empty, but an `if` statement tests the value of `isCarryingKey`. If that is true, then the message is changed to inform the user that the player is carrying a key.

www.it-ebooks.info

The `OnTriggerEnter()` method tests the `tag` string of any object the player's character controller collides with that has its **Is Trigger** checkbox set to true.

Each time the player's character controller collides with any object that has its **Is Trigger** value set to true, an `OnTriggerEnter()` event message is sent to both of the objects involved in the collision.

The `OnTriggerEnter()` message is passed a parameter, which is the `Collider` component inside the object it just collided with. Our character controller's `OnTriggerEnter()` method tests the `tag` string of the object it collided with to see if has the `key` value.

Since the **Cube–key** cube that we created has its trigger set, and has the `key` tag, then the `if` statement inside this method will detect a collision with **Cube–key** and will perform the following two actions:

- This method sets the `isCarryingKey` Boolean variable to `true`
- It also destroys the game object it has just collided with (in this case, **Cube–key**).

Boolean variables are often referred to as flags

The use of a `bool` (true/false) variable, to represent whether a particular feature of the game state is true or false, is very common. Programmers often refer to these variables as "flags". So programmers might refer to the `isCarryingKey` variable as the key carrying flag.

There's more...

Here are some details you don't want to miss:

Comparing the gameObject tag with a string special method

Comparing the `tag` string of a game object against a string is such a common task that Unity has provided a special method for just this purpose. This method is named `CompareTag()`, and takes a string as its single argument. We can therefore rewrite the `if` statement in the `OnTriggerEnter` method a little more elegantly, as follows:

```
if( hitCollider.CompareTag("key") )
```

See also...

- The *Displaying inventory icons for single object pickups* recipe.
- The *Managing inventories with a general purpose PickUp class* recipe.

www.it-ebooks.info

Displaying inventory icons for single object pickups

The previous recipe communicated whether or not the player was carrying a key via text on screen. The use of graphical icons often results in a more engaging GUI.

Getting ready

In the `0423_04_12` folder, you'll find a key icon image, and an empty inventory icon image. In the `0423_04_11` folder, you'll find some key images, and a crisscross image to apply to the terrain.

How to do it...

To display graphical icons for individual inventory objects, please follow these steps:

1. Create a new scene, and add a directional light.
2. Create a new terrain, with **Size** set to 2000 x 2000 and positioned at (-1000, 0, -1000). Now, apply a texture to it.
3. Create a cube named `Cube-key` at position (0, 1, 5) and with scale (2, 2, 2).
4. Tag `Cube-key` with the `key` string, and tick its **IsTrigger** checkbox.
5. Add a key image to `Cube-key`.
6. Import the built-in **Character Controller** Unity package and add a **3rd Person Controller** to your scene at position (0, 1, 0).
7. Attach the following C# script class to your character controller:

```
// file: PlayerInventoryIcon.cs
using UnityEngine;
using System.Collections;

public class PlayerInventoryIcon : MonoBehaviour
{
  public Texture keyIcon;
  public Texture emptyIcon;

  private bool isCarryingKey = false;

  private void OnGUI()
  {
    if( isCarryingKey )
      GUILayout.Label( keyIcon );
    else
```


www.it-ebooks.info

```
        GUILayout.Label( emptyIcon );
    }

    private void OnTriggerEnter(Collider hitCollider)
    {
      if( "key" == hitCollider.tag )
      {
        isCarryingKey = true;
        Destroy ( hitCollider.gameObject );
      }
    }
}
```

8. With the **3rd Person Controller** selected in the **Hierarchy** view, drag the key icon and empty icon images into the **Inspector** view for the corresponding public variables.

How it works...

The two public `Texture` variables hold the icons for a key and an empty inventory. The `OnGUI()` method uses an `if` statement to test the value of `isCarryingKey`. If it is true, then a key icon image is displayed, if not, then an icon representing an empty inventory is displayed. The following screenshot shows an example of the statement being true:

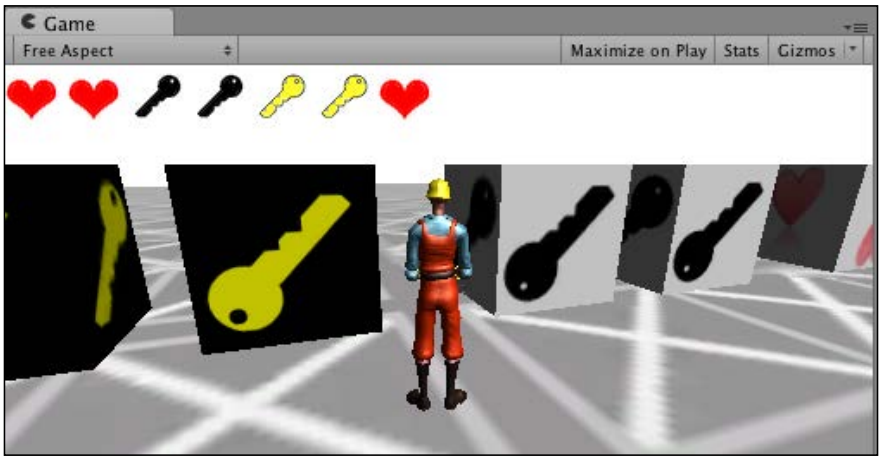
See also

- The *Displaying inventory text for single object pickups* recipe.
- The *Managing inventories with a general purpose PickUp class* recipe.

www.it-ebooks.info

Managing inventories with a general purpose PickUp class

There are often several different kinds of items players are expected to pick up and collect during a game (keys, extra lives, items that score points, and so on). A general `PickUp` class to represent the properties for each pickup item can be very useful, and the GUI display of inventory items can be made straightforward using a C# `List<T>` collection of such objects.



Getting ready

In the `0423_04_13` folder you'll find a selection of images and icons for keys and hearts.

How to do it...

To display graphical icons for a general purpose inventory, please follow these steps:

1. Create a new scene and add a directional light.
2. Create a new terrain size, with **Size** set to 2000 x 2000 and positioned at (-1000, 0, -1000). Now, apply a texture to it.
3. Create the following C# script class:

```
// file: PickUp.cs
using UnityEngine;
using System.Collections;

public class PickUp : MonoBehaviour {
  public enum PickUpCategory{
```


www.it-ebooks.info

```
    KEY, HEALTH, SCORE
  }

  public Texture icon;
  public int points;
  public string fitsLockTag;
  public PickUpCategory catgegory;
}
```

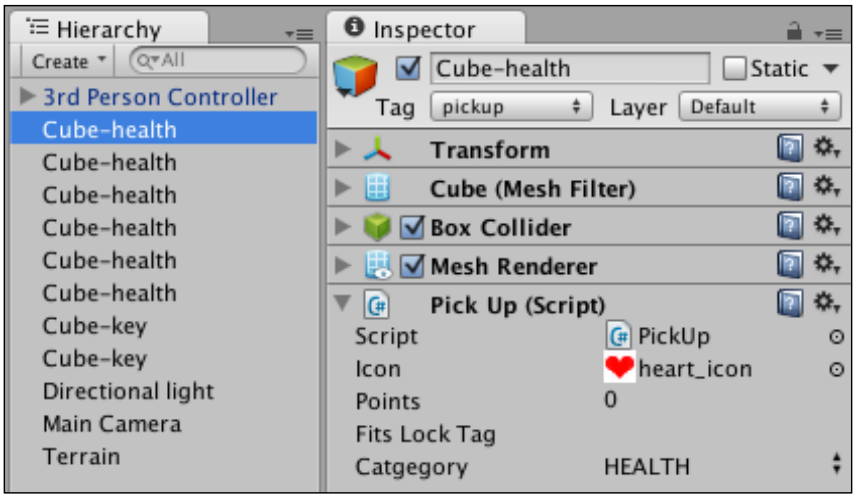
4. Import the built-in **Character Controller** Unity package and add a **3rd Person Controller** to your scene at position (0, 1, 0).
5. Attach the following C# script class to your **3rd Person Controller**:

```
// file: GeneralInventory.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class GeneralInventory : MonoBehaviour {
  const int ICON_HEIGHT = 32;
  private List<PickUp> inventory = new List<PickUp>();

  private void OnGUI(){
    // restrict display to left of screen
    Rect r = new Rect(0,0,Screen.width/2, ICON_HEIGHT);
    GUILayout.BeginArea(r);
    GUILayout.BeginHorizontal();

    DisplayInventory();

    GUILayout.FlexibleSpace();
    GUILayout.EndHorizontal();
    GUILayout.EndArea();
  }

  private void DisplayInventory(){
    foreach (PickUp item in inventory)
        GUILayout.Label( item.icon );
  }

  private void OnTriggerEnter(Collider hitCollider){
    if( "pickup" == hitCollider.tag ){
      PickUp item = hitCollider.GetComponent<PickUp>();
      inventory.Add( item );
```


www.it-ebooks.info

```
            Destroy ( hitCollider.gameObject );
        }
    }
}
```

6. Create a cube named `Cube – health` at position (0, 1, 5) with scale (2, 2, 2), and do the following (use the next screenshot as a reference):
    - Tag this objet with **pickup**
    - Tick its **Is Trigger** checkbox
    - Add the `PickUp` script class as a component of this object
    - Drag the heart icon into the **Inspector** view for the corresponding public variable
    - Choose **HEALTH** form the **Category** dropdown list
    - Add the heart image to the material for this object



7. Make several duplicates of **Cube – health**, and change some of them to have colored key images and icons as appropriate. This will ensure that there are several different items for the player to pick up in your game.

How it works...

The `PickUp` class has no methods, but it declares several useful public variables. It also declares a public `enum` category defining three possible categories of pickups: `KEY`, `HEALTH`, and `SCORE`. Each object of the `PickUp` class can define an image icon, an integer number of points, a string (for key items, the tag of locks that such a key would fit), and which of the `enum` categories each `PickUp` item belongs to.

www.it-ebooks.info

Game objects can have a scripted object of the `PickUp` added as a component. In this recipe, cubes with images to illustrate what they are have been used, but the game objects the player meets in the game world could be interactive 3D objects or whatever. Each game object with a `PickUp` component needs to have its properties set appropriately. For example, yellow key objects should have a yellow key icon assigned, be set as being in the `KEY` category, and have the **tag** string set for locks it can open.

The player's character controller has the `GeneralInventory` script added. This is a relatively straightforward GUI script that has two main functions: first, it maintains a `List<T>` collection of `PickUp` objects, representing items the player is "carrying" at any point in time, and it displays the icons for each item via the `OnGUI()` method. Second, it detects collisions with pickup game objects via the `OnTriggerEnter()` method, and objects tagged `pickup` that are collided with are added to the inventory list.

There's more...

Here are some details you don't want to miss:

Responding to non-display pickups

By having different categories of pickups, the actions for collisions can be different. So perhaps for `HEALTH` pickups, the `Points` value is added to the player's health, and the object is destroyed rather than being added to the inventory. This would simply require an `if` or a `case` statement inside the `OnTriggerEnter()` method to decide what to do once an object tagged `pickup` has been collided with.

Removing items from the List<> collection

Events may occur (for example, open a door for a key being carried) that result in an item needing to be removed from the inventory `List<>`. For example, if a yellow door were collided with, and the player was carrying a key that could open such doors, then that item should be removed from the inventory `List<>` collection and the door be opened. To implement this, you would need to add an `if` statement test inside `OnTriggerEnter()` to detect a collision with the item tagged `yellowDoor`:

```
if( "yellowDoor" == hitCollider.tag )
  OpenDoor(hitCollider.gameObject);
```

The `OpenDoor()` method would need to identify which item (if any) in the inventory can open such a door, and if found, then that item should be removed from the `List<>` collection and the door be opened by the appropriate method:

```
private void OpenDoor(GameObject doorGO){
  // search for key to open the tag of doorGO
  int colorKeyIndex = FindItemIndex(doorGO.tag);
  if( colorKeyIndex > -1 ){
```


www.it-ebooks.info

```
        // remove key item
        inventory.RemoveAt( colorKeyIndex );

        // now open the door ...
        doorGO.animation.Play ("open");
      }
    }

    private int FindItemIndex(string doorTag){
      for (int i = 0; i < inventory.Count; i++){
        PickUp item = inventory[i];
        if( item.fitsLockTag == doorTag )
          return i;
      }
    }
```

See also

- The *Displaying inventory text for single object pickups* recipe.
- The *Displaying inventory icons for single object pickups* recipe.

Controlling the scrollbar with the mouse wheel

For many users, there is nothing more natural than moving a scrollbar with the mouse wheel. However, this is not natively supported by Unity. In this recipe, we will add mouse wheel support to a vertical scrollbar.

How to do it...

To control the scrollbar with the mouse wheel, please follow these steps:

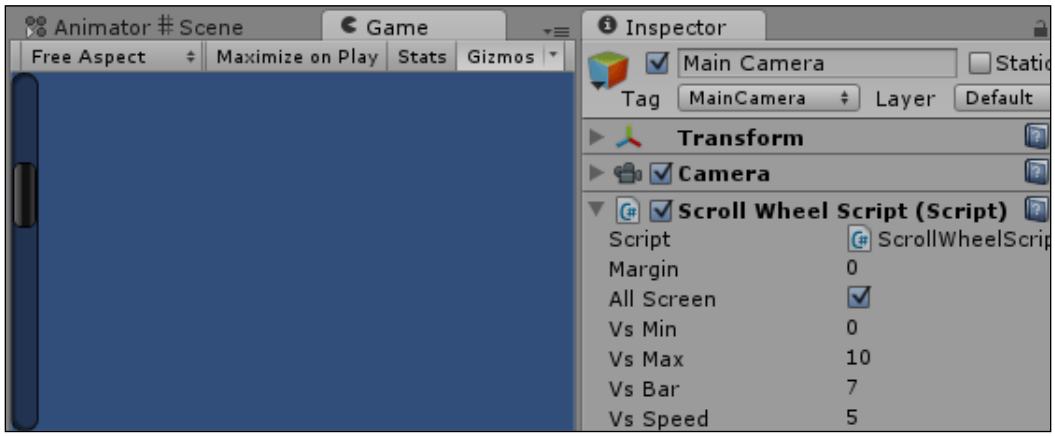
1. Attach the following C# script class to the **Main Camera**:

```
// file: ScrollWheel.cs
using UnityEngine;
using System.Collections;

public class ScrollWheel : MonoBehaviour {
    public int margin;
    public bool allScreen = true;
    public float min = 0.0f;
```


www.it-ebooks.info

```
    public float max = 10.0f;
    public float value = 10.0f;
    public float speed = 5;

    void OnGUI() {
        Rect rect = new Rect(margin, margin, 30, Screen.height -
(margin * 2));
        value = GUI.VerticalScrollbar(rect, value, 1.0f, max,
min);
        bool onArea;
        if (!allScreen && !rect.Contains(Input.mousePosition)) {
            onArea = false;
        }
        else {
            onArea = true;
        }

        float vsMove = Input.GetAxis("Mouse ScrollWheel") * speed;
        if ((value + vsMove) > min && onArea) {
            value += vsMove;
        }
    }
}
```

2. Test your scene. You should now be able to control the scrollbar with the mouse wheel.



www.it-ebooks.info

How it works...

Since the scrollbar's handle position is given by a variable (in our case, `vsBar`), all we needed to do was increment that variable according to the mouse wheel input.

We have also added a few variables to enhance the scrollbar customization: margin, minimum and maximum values, scroll speed; and also a Boolean variable to enable/disable the mouse wheel input when the mouse cursor is not directly on top of the scrollbar.

There's more...

In this recipe, the scrollbar height is determined by the application screen's height. However, there's no reason you couldn't add more variables to manually set its size and position.

Implementing custom mouse cursor icons

Cursor icons are often used to indicate the nature of the interaction that can be done with the mouse. Zooming, for instance, might be illustrated by a magnifying glass. Shooting, on the other hand, is usually represented by a stylized target. In this recipe, we will learn how to implement custom mouse cursor icons to better illustrate your gameplay—or just to escape the Windows and OS X default GUI.

Getting ready

If you need a set of textures to be used as cursor icons, please use the three image files available in the `0423_04_15` folder.

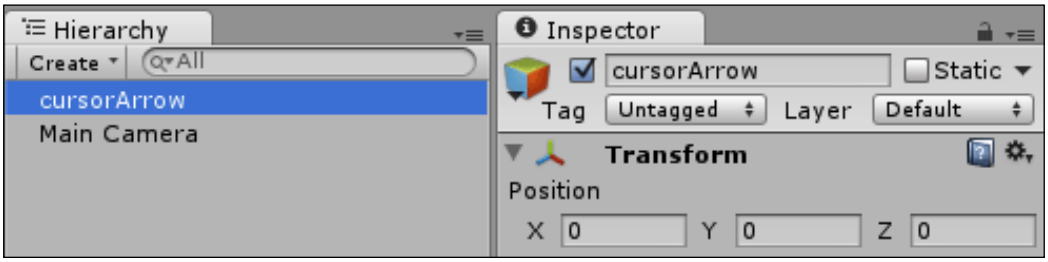
How to do it...

To implement custom mouse cursors, please follow these steps:

1. Import the textures into your Unity project.
2. In the **Project** window, select the **cursorArrow** texture.
3. Create a new GUI texture by navigating to **GameObject** | **Create Other** | **GUI Texture**. Since you had the **cursorArrow** texture selected, the **GUI Texture** will use it.

www.it-ebooks.info

4. In the **Transform** component of the **Inspector** view, set the GUI texture's **Position** to (0,0,0), as shown in the following screenshot:



5. Create a new C# script class named `CursorScript` containing the following code:

```
// file: CursorScript.cs
using UnityEngine;
using System.Collections;

public class CursorScript : MonoBehaviour
{
    public Texture2D iconArrow;
    public Vector2 arrowRegPoint;
    public Texture2D iconZoom;
    public Vector2 zoomRegPoint;
    public Texture2D iconTarget;
    public Vector2 targetRegPoint;
    private Vector2 mouseReg;

    void Start()
    {
        guiTexture.enabled = true;
        if (iconArrow)
        {
            guiTexture.texture = iconArrow;
            mouseReg = arrowRegPoint;
            Screen.showCursor = false;
        }
    }

    void Update()
    {
        Vector2 mouseCoord = Input.mousePosition;
        Texture mouseTex = guiTexture.texture;
```


www.it-ebooks.info

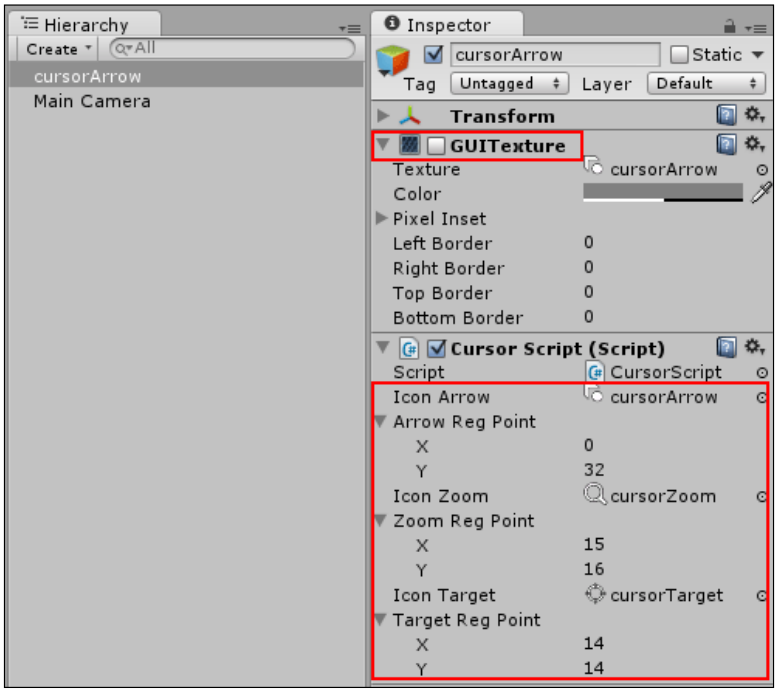
```
        guiTexture.pixelInset = new Rect(mouseCoord.x -
(mouseReg.x), mouseCoord.y - (mouseReg.y), mouseTex.width,
mouseTex.height);

        if (Input.GetKey(KeyCode.RightShift) || Input.
GetKey(KeyCode.LeftShift))
        {
            if (iconTarget)
            {
                guiTexture.texture = iconTarget;
                mouseReg = targetRegPoint;
            }
        }
        else if (Input.GetMouseButton(1))
        {
            if (iconZoom)
            {
                guiTexture.texture = iconZoom;
                mouseReg = zoomRegPoint;
            }
        }
        else
        {
            if (iconArrow)
            {
                guiTexture.texture = iconArrow;
                mouseReg = arrowRegPoint;
            }
        }
    }

}
```

6. Save your script and apply it to the GUI texture you have created by dragging it from the **Project** window to the **GUI Texture** game object in the **Hierarchy** window.
7. In the **Hierarchy** window, select the GUI texture and disable its **GUITexture** component.
8. Drag the **cursorArrow**, **cursorZoom**, and **cursorTarget** texture files to the **Icon Arrow**, **Icon Zoom**, and **Icon Target** fields respectively.

www.it-ebooks.info

9. Change the **Arrow Reg Point**, **Zoom Reg Point**, and **Target Reg Point** fields to **X:0 Y:32**, **X:15 Y:16**, and **X:14 Y:14** respectively (use the following screenshot for your reference):

The **Reg Point** values are actually the pixel coordinates for the focus point of the cursor. When in doubt, open your image editor and check it.



10. Play your scene. Your operating system's default cursor should be replaced by the **cursorArrow** texture; changed to `cursorZoom` when the right mouse button is clicked and to `cursorTarget` when any *Shift* key is down.

How it works...

The script updates the GUI texture's position based on the mouse cursor position and registration points for each cursor type. The texture map is chosen according to the user's action (pressing the *Shift* key or clicking the right mouse button will change the texture). The script also hides the original mouse cursor, so it won't be on top of your custom one.

www.it-ebooks.info

www.it-ebooks.info

5
Controlling Animations

In this chapter, we will cover:

- Configuring a character's Avatar and idle animation
- Moving your character with Root Motion and Blend Trees
- Mixing animations with Layers and Masks
- Overriding Root Motion via script
- Adding rigid props to animated characters
- Making an animated character throw an object
- Applying Ragdoll physics to a character
- Rotating the character's torso to aim

Introduction

Arguably the most celebrated feature in Unity 4, the **Mecanim** animation system is indeed a revolution in how characters are animated and controlled within the engine. In this chapter, we will learn how to take advantage of its flexibility, power, and friendly and highly visual interface.

All the recipes that deal with animated characters make use of **Mixamo**'s suite of characters, motion packs, and scripts. Mixamo is a complete solution for character rigging and animation, and it can even provide you with rigged 3D characters at low or no cost. You can find out more about it at Unity's Asset Store at `u3d.as/content/mixamo/mixamo-animation-store/1At` or on their website at `www.mixamo.com`.

www.it-ebooks.info

Configuring a character's Avatar and Idle animation

One of the features that make Unity's new animation system, Mecanim, so flexible and powerful is the ability to quickly reassign animation clips from one character to another. This is made possible through the use of **Avatars**; an Avatar is basically a layer between your character's original rig and Unity's animation system.

In this recipe, we will learn how to configure an Avatar skeleton on a Mixamo character.

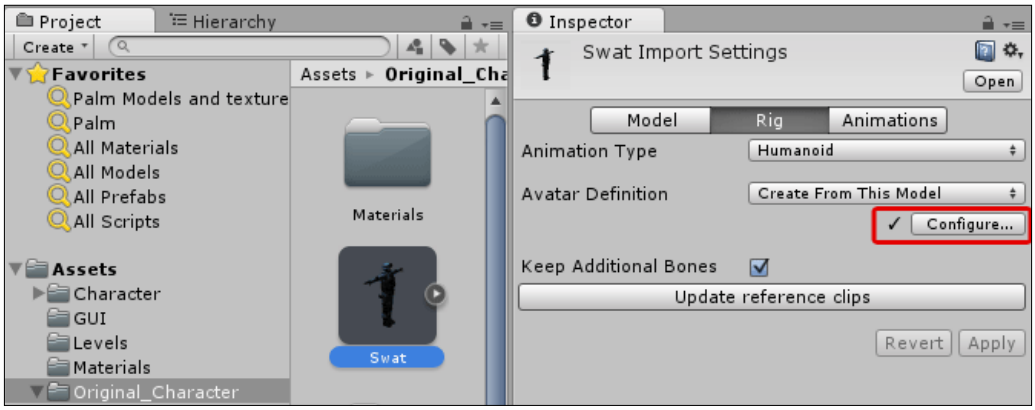
Getting ready

For this recipe, we have prepared a project named `MixamoProject`, containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

How to do it...

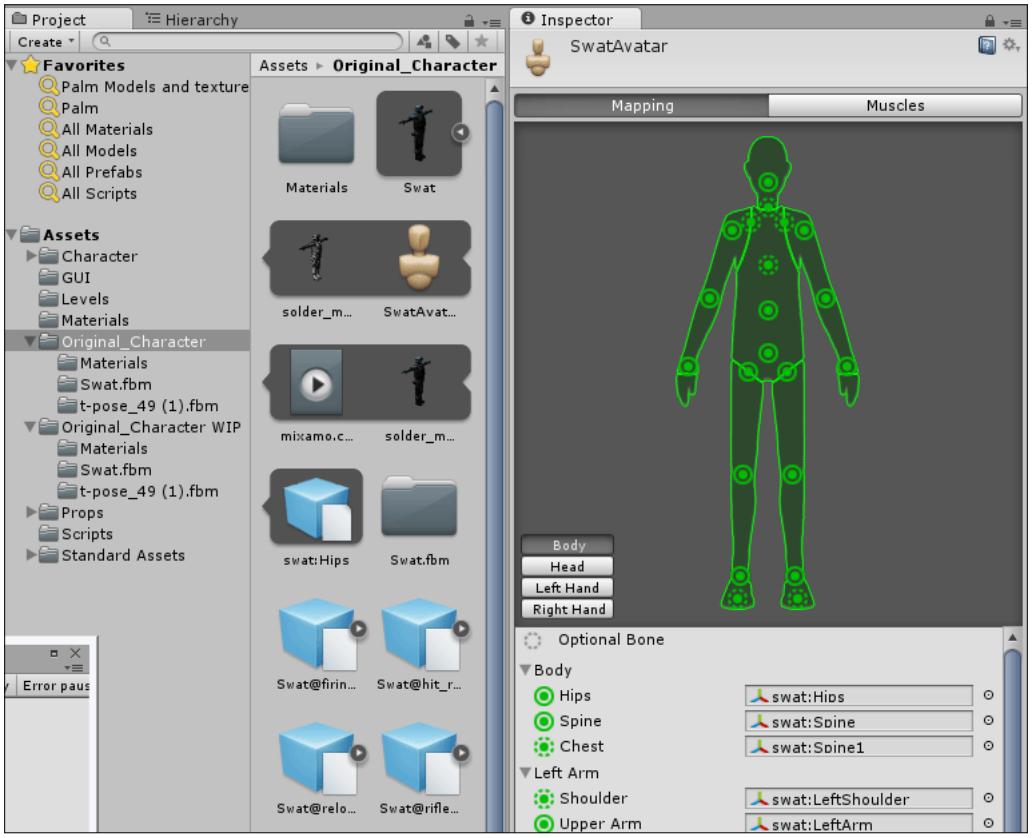
To configure an Avatar skeleton, perform the following steps:

1. Open the project, and then open the level named **05_01** (inside the folder **Levels**).
2. In the **Project** view, inside the **Original_Character** folder, select the **Swat** model.
3. In the **Inspector** view, under **Swat Import Settings**, click on the **Rig** tab. Then, change **Animation Type** to **Humanoid**. Leave **Avatar Definition** as **Create From this Model**, and also make sure that the option **Keep Additional Bones** is selected. Finally, click on the **Configure...** button.



www.it-ebooks.info

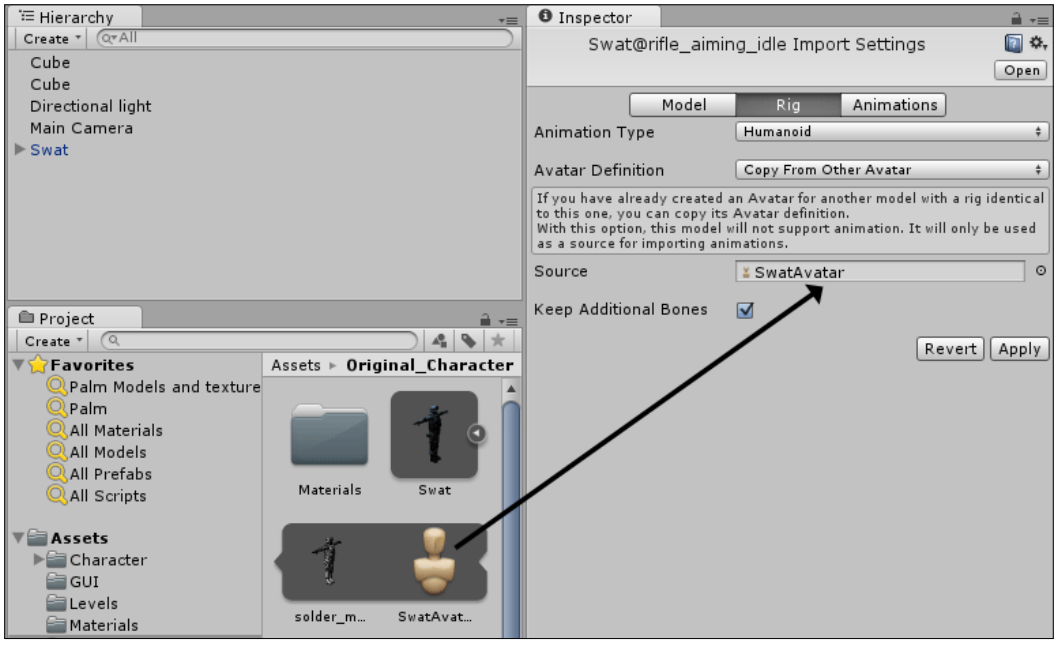
4. The **Inspector** view will show the newly-created Avatar. Observe how Unity correctly maps the bones of our character onto its structure, assigning (for example) the **swat:LeftForeArm** bone as the Avatar's **Lower Arm**. We could, of course, reassign the bones if needed. For now, just click on the **Done** button to close the view.



5. Now that we have our Avatar ready, let's configure our animation for the Idle state. From the **Original_Character** folder, select the **Swat@rifle_aiming_idle** file.

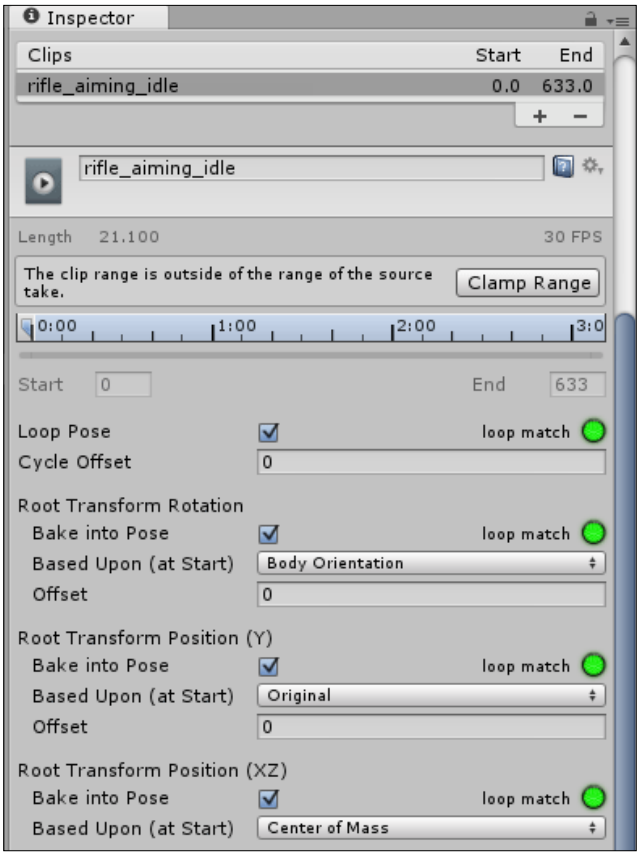
www.it-ebooks.info

6. Activate the **Rig** section, change **Animation Type** to **Humanoid** and **Avatar Definition** to **Copy From Other Avatar**. Then, select **swatAvatar** for **Source**, by choosing it from the list or dragging it into the slot from the **Project** view. Confirm your changes by clicking on **Apply**.



7. Now click on the **Animations** tab. Select the **rifle_aiming_idle** clip from the **Clips** list and select the **Loop Pose** option. Under **Root Transform Rotation**, check **Bake into Pose** and set **Body Orientation** for **Baked Upon (at Start)**; Under **Root Transform Position (Y)**, check **Bake into Pose** and set **Original** for **Baked Upon (at Start)**; **Under Root Transform Position (XZ)**, check **Bake into Pose** and set **Center of Mass** for **Baked Upon (at Start)**. Finally, click on the **Clamp Range** button (see the following screenshot) to adjust the timeline. Click on **Apply** to confirm changes.

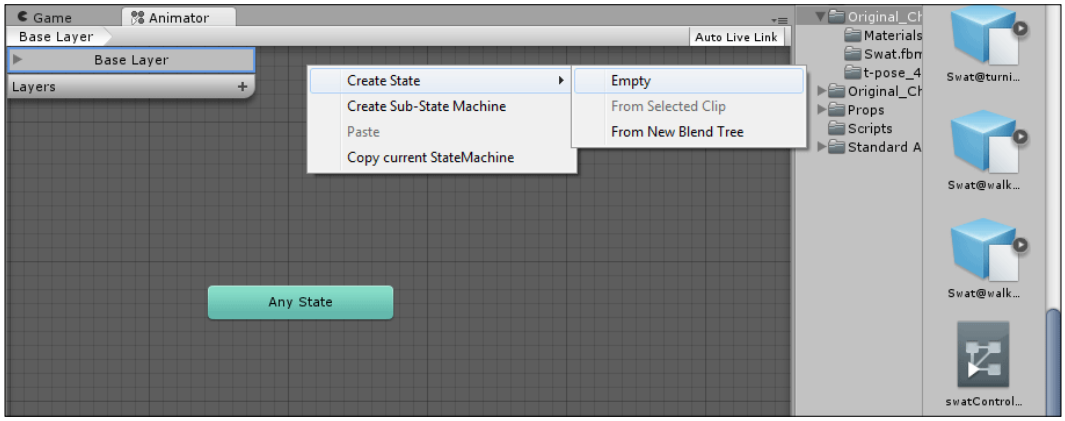
www.it-ebooks.info



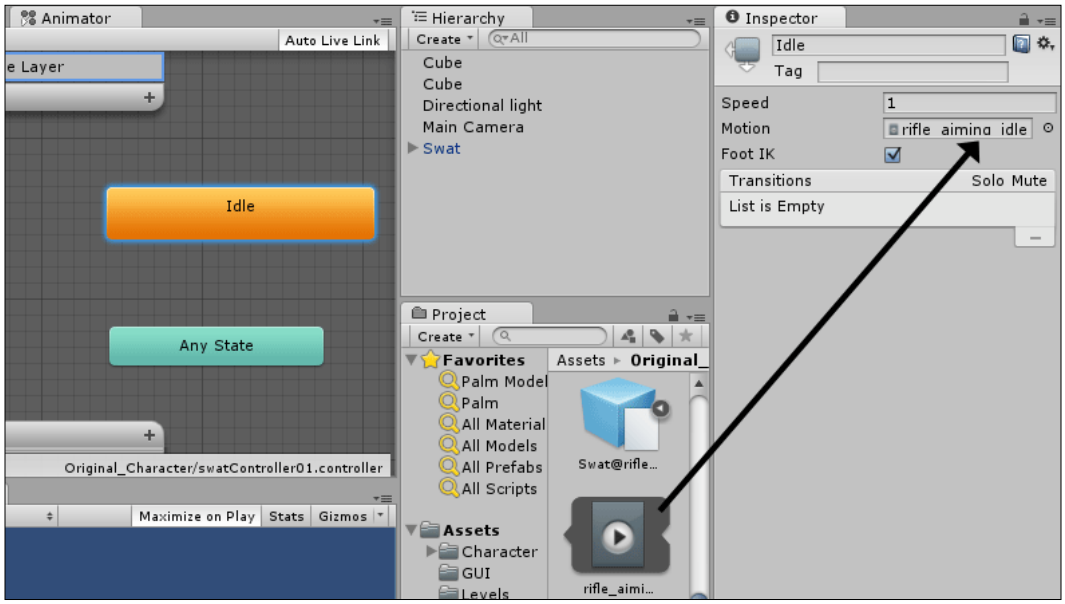
8. In order to access animation clips and play them, we need to create a controller. Do that by clicking on the **Create** button from the **Project** view and selecting the option **Animator Controller**. Name it `swatController01`.
9. Double-click on **Animator Controller** to open the **Animator** view.

www.it-ebooks.info

10. In the **Animator** view, right-click on the grid to open a context menu. Then, select the **Empty** option under **Create State** (see the following screenshot). A new box named **New State** will appear, which is orange in color, indicating that this is the default state.



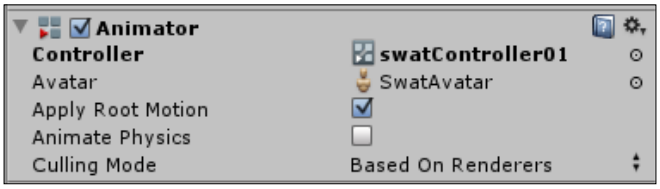
11. Select this box and, in the **Inspector** view, change its name to `Idle`. Also, for the **Motion** field, choose **rifle_aiming_idle** by either selecting it from the list or dragging it from the **Project** view.



12. Drag the **Swat** model from the **Original_Character** folder onto the **Hierarchy** view, placing it into the scene.

www.it-ebooks.info

13. Select the **Swat** model from the **Hierarchy** view and observe its Animator component in the **Inspector** view. Then, assign the newly created **swatController01** to the **Controller** field.



14. Play your scene to see the character correctly animated.

How it works...

As you might have noticed, we performed a variety of operations in order to have our character configured and animated in the scene. First, we set up its Avatar skeleton based on its original bone structure. Then, we applied the character's Avatar in the animation clip (which, like the character itself, is a `.fbx` file). After that, we adjusted the animation clip to better suit our character, which included making it loop. Finally, an Animator Controller was created, and the edited animation clip was made into its default animation state.

The concept of the Avatar is what makes Mecanim so flexible. Once you have a Controller, you can apply it to other humanoid characters as long as they have an Avatar body mask. If you want to try it yourself, import the `mascot.fbx` file (located inside the `0423_05_01` folder), perform steps 3 and 4 of this recipe for that character, place it on the scene, and use **swatController01** as its **Controller** in the Animator component. Then, play the scene to see the mascot playing the **rifle_aiming_idle** animation clip.

There's more...

To get more information about the Animator Controller, check Unity's documentation at `http:// http://docs.unity3d.com/Documentation/Components/class-AnimatorController.html`.

Moving your character with Root Motion and Blend Trees

The Mecanim animation system is capable of applying Root Motion to characters. In other words, it actually moves the character according to the animation clip, as opposed to arbitrarily translating the character model while playing an in-place animation cycle. This makes most Mixamo animation clips perfect for use with Mecanim.

www.it-ebooks.info

Another new feature for the animation system is Blend Trees, which can make transitions between animation states smooth and easy. In this recipe, we will take advantage of these new features to make our character walk and run forwards or backwards, and also strafe right and left, at different speeds.

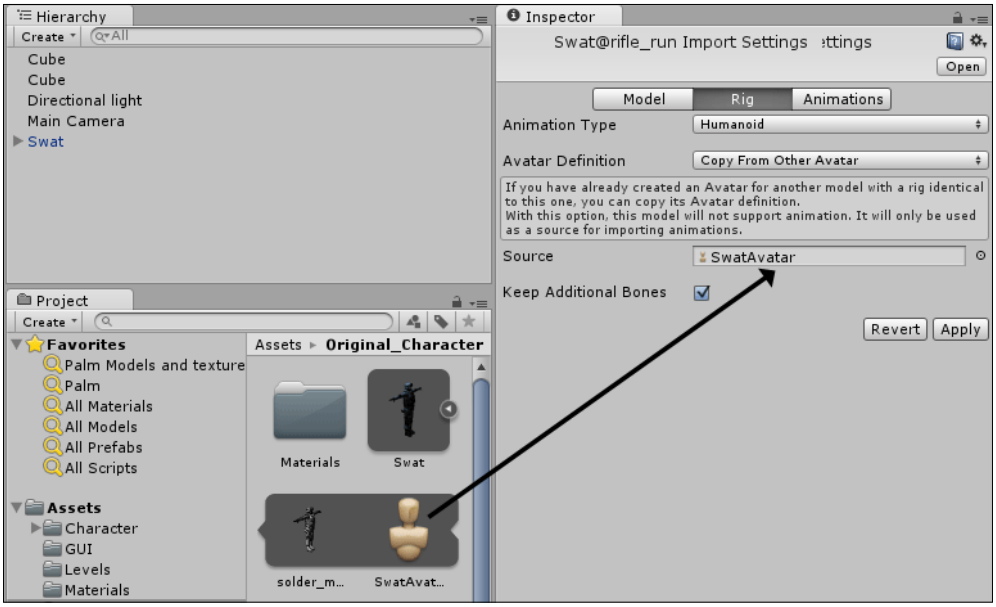
Getting ready

For this recipe, we have prepared a project named `MixamoProject`, containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

How to do it...

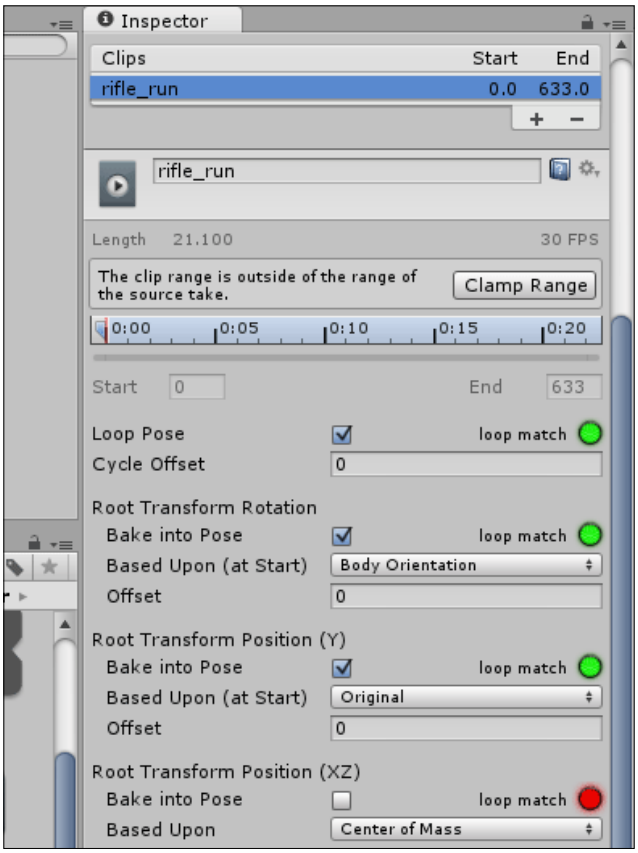
To apply Root Motion to your character using Blend Trees, perform the following steps:

1. Open the project, and then open the level named **05_02** (inside the **Levels** folder). Note that it includes a S.W.A.T. character that features the Animator component using the **swatController02** Controller.
2. We need to configure our animation clips. From the **Original_Character** folder, select the **Swat@rifle_run** file.
3. Click on the **Rig** tab. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Copy From Other Avatar**. Then, select **swatAvatar** for **Source** by choosing it from the list or dragging it from the **Project** view into the slot. Confirm your changes by clicking on **Apply**.



www.it-ebooks.info

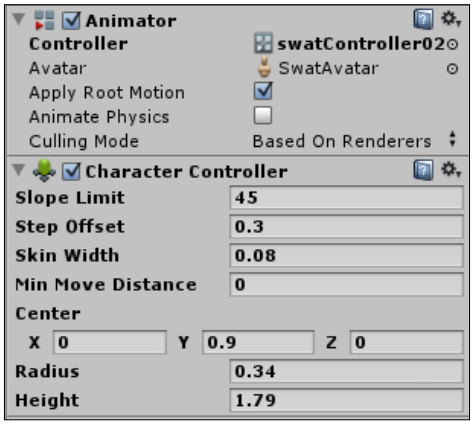
4. Now click on the **Animations** tab. Select the clip **rifle_run** (from the **Clips** list) and select the **Loop Pose** option. Under **Root Transform Rotation**, check **Bake into Pose** and set **Body Orientation** for **Baked Upon (at Start)**; Under **Root Transform Position (Y)**, check **Bake into Pose** and set **Original** for **Baked Upon (at Start)**; Under **Root Transform Position (XZ)**, leave the **Bake into Pose** checkbox deselected. Finally, click on the **Clamp Range** button to adjust the timeline (shown in the following screenshot). Click on **Apply** to confirm changes.



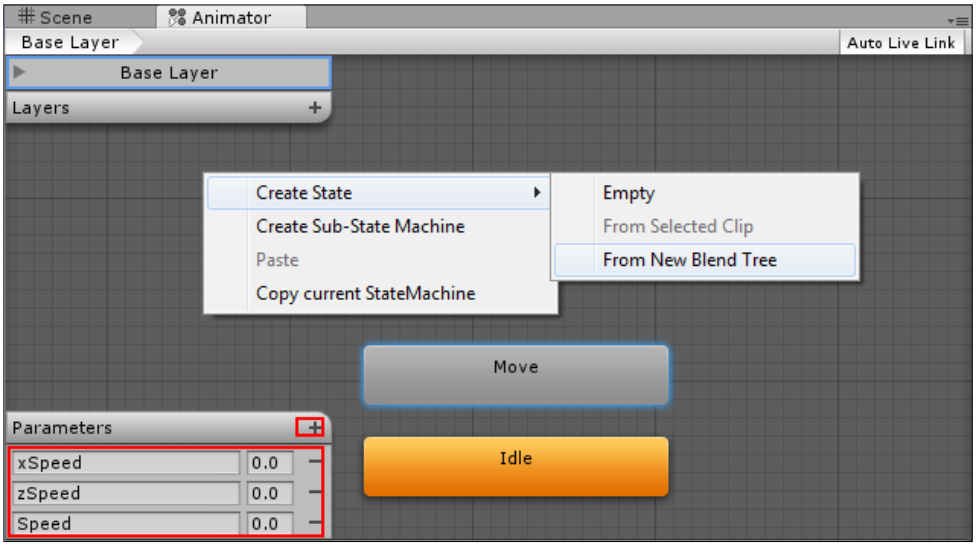
5. Repeat steps 3 and 4 for each one of the following animation clips inside the **Original_Character** folder: **Swat@run_backwards**, **Swat@strafe**, **Swat@strafe_2**, **Swat@strafe_left**, **Swat@strafe_right**, **Swat@walking**, and **Swat@walking_backwards**.

www.it-ebooks.info

6. From the **Hierarchy** view, select the **Swat** game object and attach a Character Controller component to it by going to **Component** | **Physics** | **Character Controller**. Then, set the following coordinates for **Center**: `(0, 0.9, 0)`; change the entry in the **Radius** field to `0.34` and the **Height** field to `1.79`:

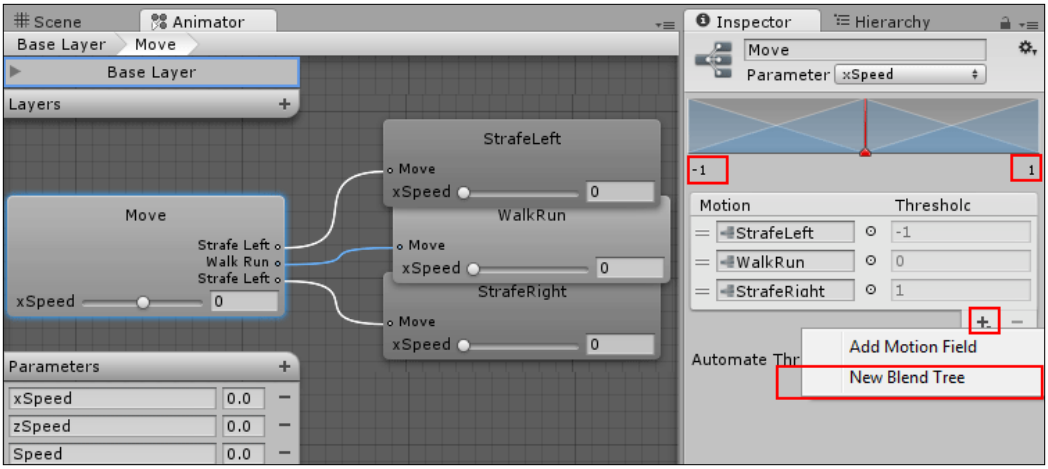


7. In the **Project** view, open the **swatController02** Controller.
8. In the bottom-left corner of the **Animator** view, create three new parameters (Float values) named `xSpeed`, `zSpeed`, and `Speed`.
9. Also, right-click on the gridded area and, from the context menu, select **From New Blend Tree** under **Create State**. Change its name to **Move** in the **Inspector** view.

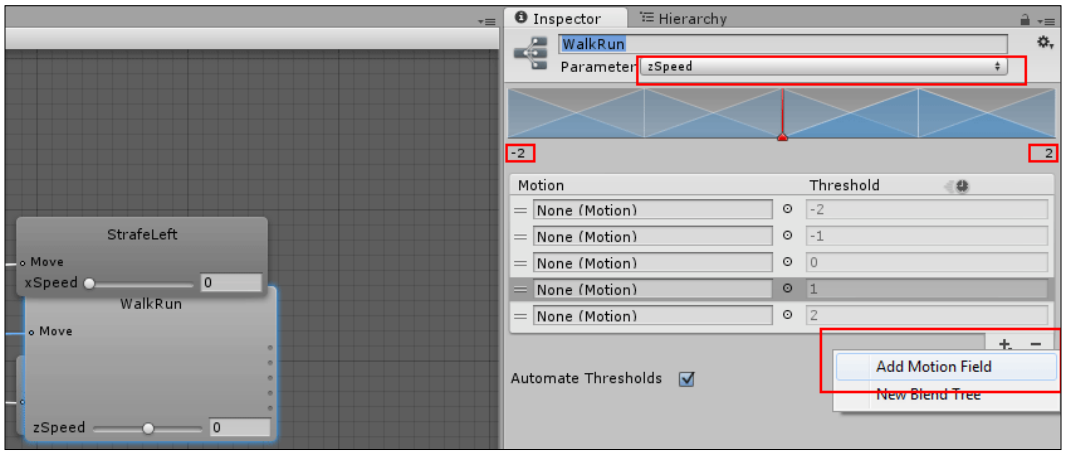


www.it-ebooks.info

10. Double-click on the **Move** state. You will see the empty Blend Tree you have created. Select it and rename it to **Move** in the **Inspector** view.
11. In the **Inspector** view, below the diagram on the top, leave the parameter in the drop-down menu as **xSpeed**, but change the parameter values to `-1` and `1`. Also, use the button with the **+** sign to add three new Blend Trees. From top to bottom, rename them as `StrafeLeft`, `WalkRun`, and `StrafeRight`.

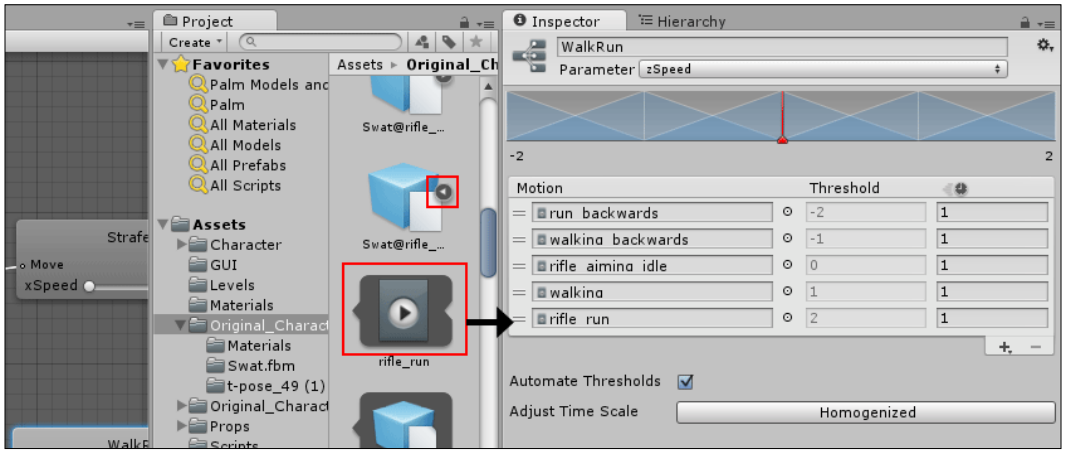


12. Select the **WalkRun** node. In the **Inspector** view, change the parameter to **zSpeed**. Then, change the parameter values to `-2` and `2`, and use the button with the **+** sign to add five Motion fields.



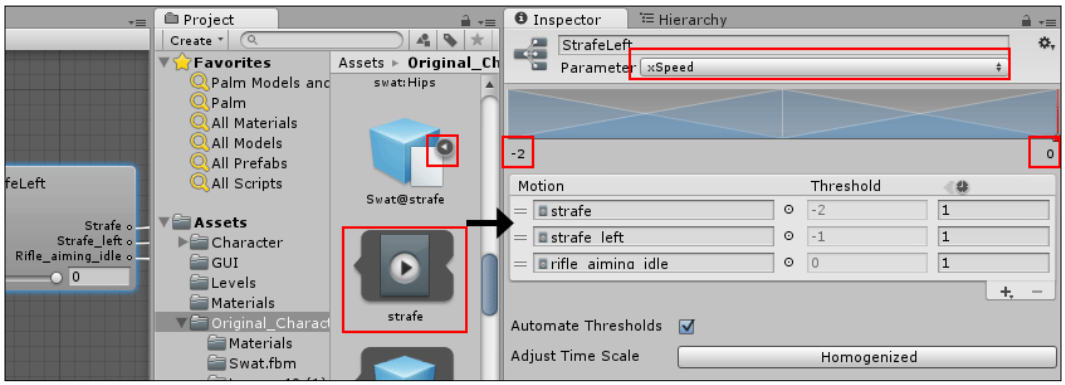
www.it-ebooks.info

13. Now, fill the Motion list with the following clips (from top to bottom): **run_backwards**, **walking_backwards**, **rifle_aiming_idle**, **walking**, and **rifle_run**. You can do this either by selecting them from the list, or if there is more than one clip with the same name, dragging them from the **Project** view into the slot (by expanding the appropriate model icon).



14. Select the **StrafeLeft** node. In the **Inspector** view, keep its parameter as **xSpeed** but change the parameter values to `-2` and `0`; use the button with the **+** sign to add three Motion fields.

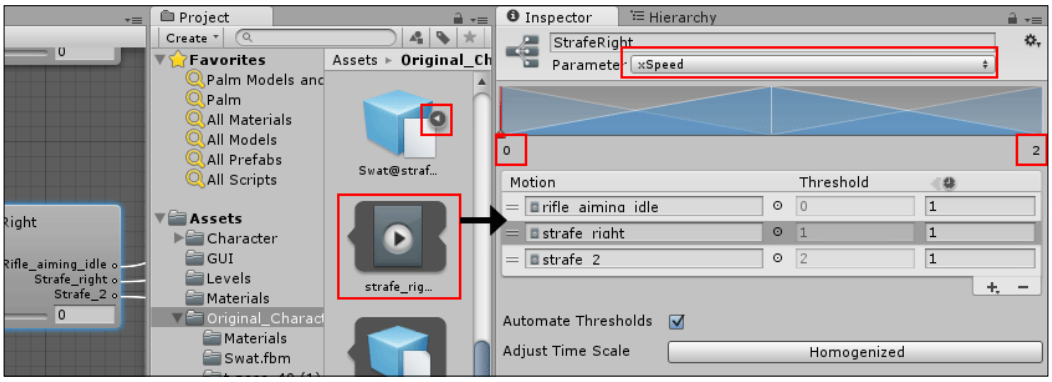
15. Fill the Motion list with the following clips (from top to bottom): **strafe**, **strafe_left**, and **rifle_aiming_idle**.



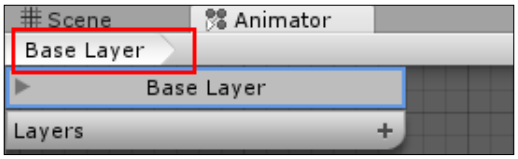
16. Select the **StrafeRight** node. In the **Inspector** view, keep its parameter as **xSpeed** but change the parameter values to `0` and `2`; use the button with the **+** sign to add three Motion fields.

www.it-ebooks.info

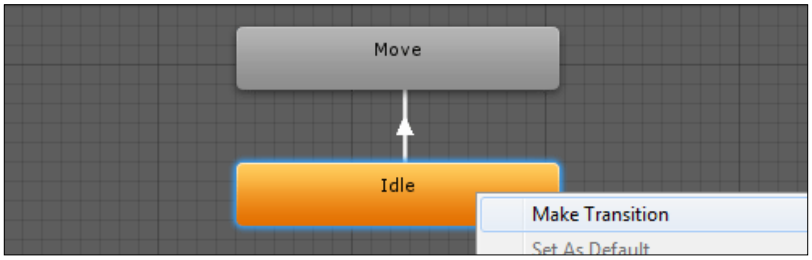
17. Fill the Motion list with the following clips (from top to bottom): **rifle_aiming_idle**, **strafe_right**, and **strafe_2**.



18. Let's go back to the base layer. In the **Animator** view, double-click on the gridded area or click on the **Base Layer** tab.



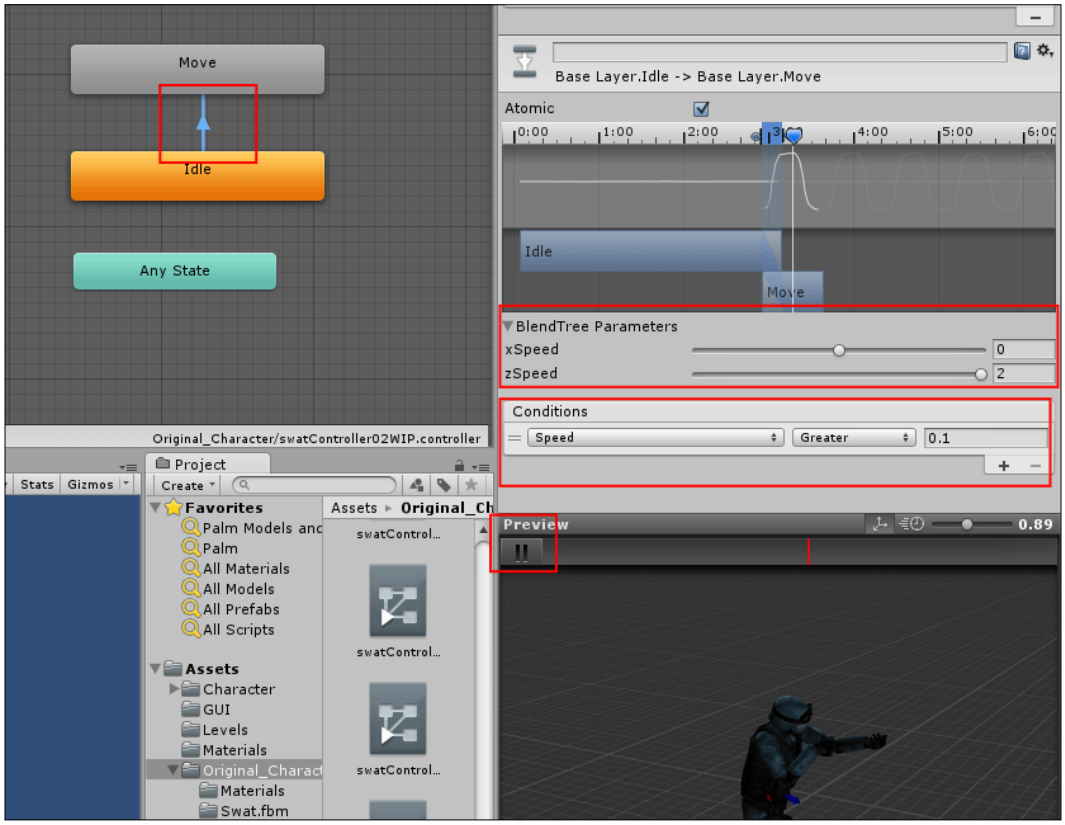
19. Right-click on the **Idle** state box and select **Make Transition** from the menu. Then, drag the white arrow into the **Move** box.



20. Select the arrow (it should turn blue). In the **Inspector** view, we can now configure the conditions that determine when and how the transition from **Idle** to **Move** is made.

www.it-ebooks.info

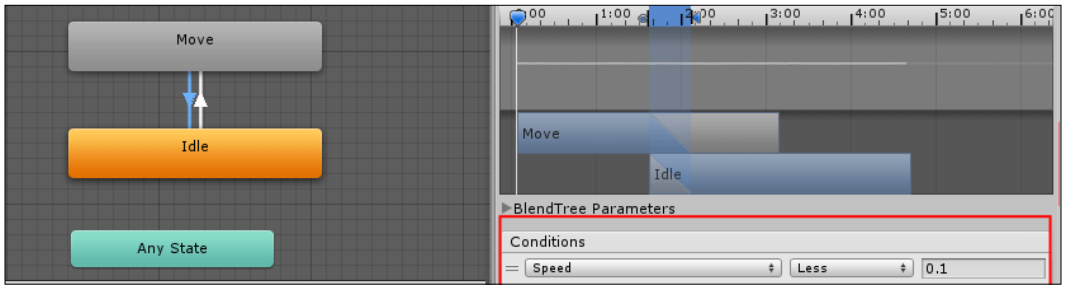
21. In the **Conditions** box, select the options **Speed**, **Greater**, and **0.1**. Then, expand the **BlendTree Parameters** section, change **zSpeed** to `2` and click on the play icon in the **Preview** window. That should give you an idea on how the transition will be made.



You can experiment with narrowing and widening the blue segment of the timeline by using the handles to get a smoother transition.

22. Now create a transition from the **Move** box to the **Idle** box. Then, select the arrow representing it.
23. In the **Conditions** box, change the parameters to **Speed**, **Less**, and **0.1** (as shown in the following screenshot):

www.it-ebooks.info



24. Now that we have established our animation states and transitions, we must create the script that will actually transform the player's input into those variables created to control the animation.
25. From the **Project** view, create a new C# script and name it `BasicController02`.
26. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class BasicController02 : MonoBehaviour {
    private Animator animator;
    private CharacterController controller;
    public float transitionTime = 0.25f;

    void Start () {
        controller = GetComponent<CharacterController>();
        animator = GetComponent<Animator>();
        if(animator.layerCount >= 2)
            animator.SetLayerWeight(1, 1);
    }

    void Update () {
        float accelerator = 1.0f;
        if(controller.isGrounded){
            if (Input.GetKey (KeyCode.RightShift) ||Input.GetKey
(KeyCode.LeftShift) ){
                accelerator = 2.0f;
            } else if(Input.GetKey (KeyCode.RightAlt) ||Input.
GetKey (KeyCode.LeftAlt) ){
                accelerator = 1.5f;
            } else {
                accelerator = 1.0f;
            }
            float h = Input.GetAxis("Horizontal");
```


www.it-ebooks.info

```
            float v = Input.GetAxis("Vertical");
            float xSpeed = h * accelerator;
            float zSpeed = v * accelerator;
            animator.SetFloat("xSpeed", xSpeed, transitionTime,
Time.deltaTime);
            animator.SetFloat("zSpeed", zSpeed, transitionTime,
Time.deltaTime);
            animator.SetFloat("Speed", Mathf.Sqrt(h*h+v*v),
transitionTime, Time.deltaTime);
            //transform.Rotate(Vector3.up * (Time.deltaTime * v *
Input.GetAxis("Mouse X") * 90), Space.World);
        }
    }
}
```

27. Save your script and attach it to the **Swat** game object in the **Hierarchy** view.
28. Play your scene and test the game. You should be able to control your character with the arrow keys (or the *W*, *A*, *S*, and *D* keys). You should also be able to get your character to run by pressing the `Shift` key or walk faster by pressing *Alt*.

How it works...

Our script will detect the user's keyboard input and translate it into variables to be passed to the Animator Controller as its parameters **Speed**, **xSpeed**, and **vSpeed**. These parameters will determine which animation clips should be played. For instance, if the player presses the down arrow and *Shift* keys together, the parameters **Speed**, **xSpeed**, and **vSpeed** will be passed with the values `1`, `0`, and `-2` respectively. A positive value for **Speed** will trigger the transition from the **Idle** state to the **Move** state. Furthermore, an **xSpeed** value of `0`, indicating that the left and right arrow keys are not being pressed, will favor the **WalkRun** Blend Tree and not **StrafeLeft** and **StrafeRight**. Finally, the **vSpeed** parameter (with a value of `-2`) will play the **walking_backwards** motion clip within the **WalkRun** Blend Tree. The character will then move according to the animation being played. Observe how pressing the *Alt* key actually blends the animation clips for walking and running in real time.

www.it-ebooks.info

There's more...

If you want to learn more about Mecanim's animation system, there are some links you might want to check out, such as Unity's documentation at `http://docs.unity3d.com/Documentation/Manual/MecanimAnimationSystem.html`.

You can also check out Mecanim Example Scenes, available at the Unity Asset Store at `http://u3d.as/content/unity-technologies/mecanim-example-scenes/3Bs` and the Mecanim video tutorials, at `http://video.unity3d.com/video/7362044/unity-40-mecanim-animation`.

Mixing animations with Layers and Masks

Mixing animations is a great way of adding complexity to your animated characters without requiring a vast number of animated clips. Using **Layers** and **Masks**, we can combine different animations by playing specific clips for specific body parts of the character. In this recipe, we will apply this technique to our animated character, triggering animation clips for firing a rifle and throwing a grenade with the character's upper body, while keeping the lower body moving or idle, according to the player's input.

Getting ready

For this recipe, we have prepared a project named `MixamoProject`, containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

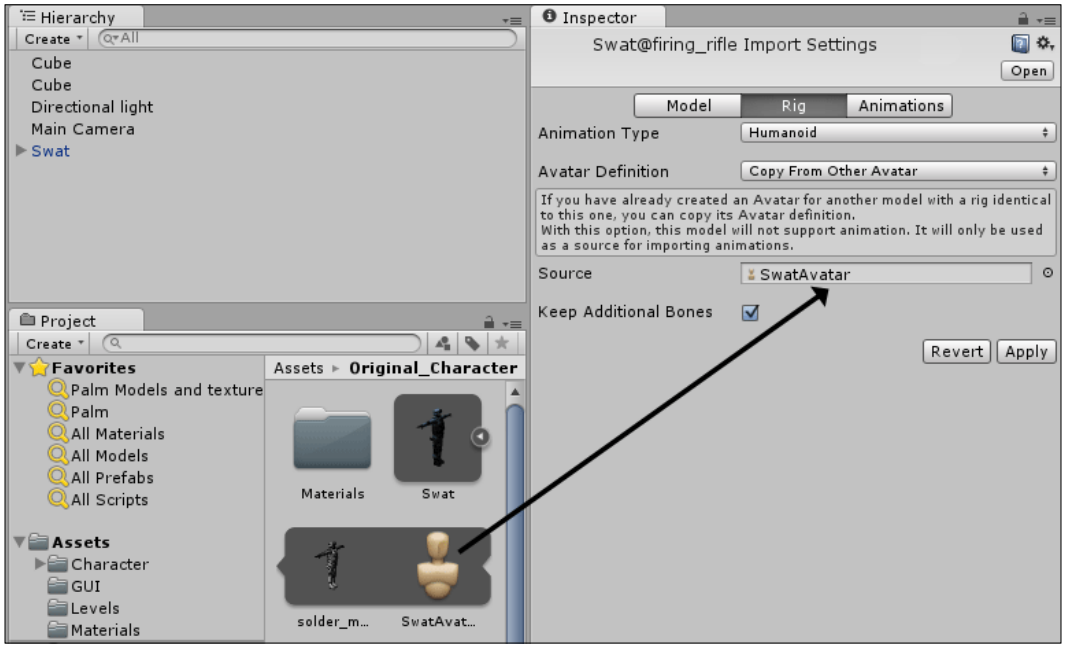
How to do it...

To mix animations using Layers and Masks, perform the following steps:

1. Open the project, and then open the level named **05_03** (inside the **Levels** folder). Note that it includes a S.W.A.T. character that features the Animator component using the **swatController03** Controller.
2. We need to configure our animation clips. From the **Original_Character** folder, select the **Swat@firing_rifle** file.

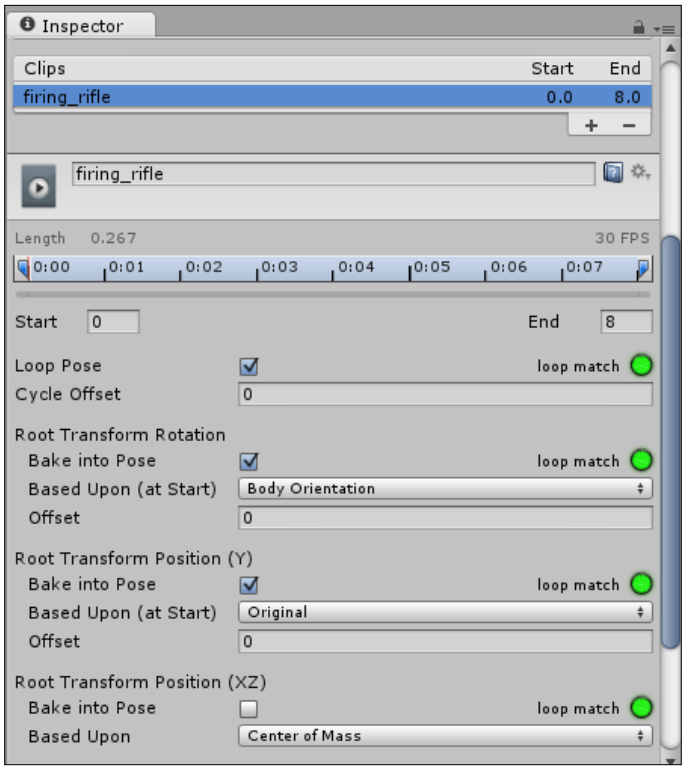
www.it-ebooks.info

3. Click on the **Rig** tab. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Copy From Other Avatar**. Then, select **swatAvatar** for **Source** by choosing it from the list or dragging it from the **Project** view into the slot. Confirm your changes by clicking on **Apply**.



4. Now click on the **Animations** tab. Select the **firing_rifle** clip from the **Clips** list and select the **Loop Pose** checkbox. Under **Root Transform Rotation**, check **Bake into Pose** and set **Body Orientation** for **Baked Upon (at Start)**; Under **Root Transform Position (Y)**, check **Bake into Pose** and set **Original** for **Baked Upon (at Start)**; Under **Root Transform Position (XZ)**, leave the **Bake into Pose** checkbox deselected. Finally, click on the **Clamp Range** button to adjust the timeline (see the following screenshot). Click on **Apply** to confirm the changes.

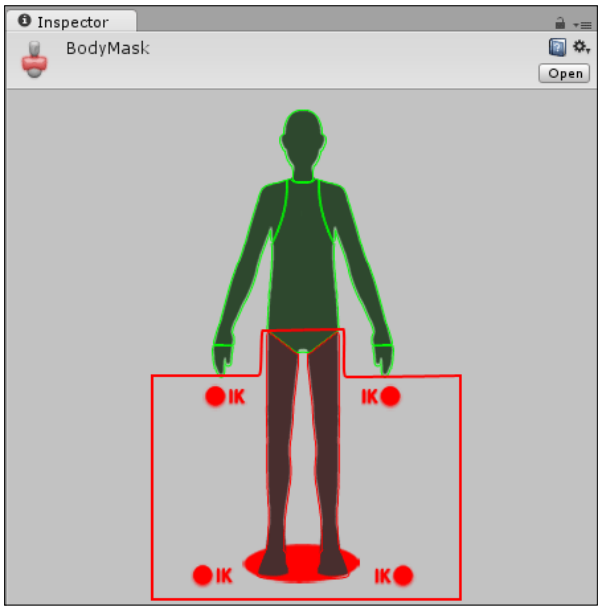
www.it-ebooks.info



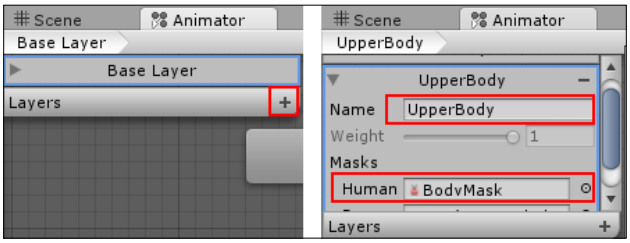
5. Now, from the **Original_Character** folder, select the **Swat@toss_grenade** file. Click on the **Rig** tab. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Copy From Other Avatar**. Then, select **swatAvatar** for **Source** by choosing it from the list or dragging it from the **Project** view into the slot. Confirm the changes by clicking on **Apply**.
6. Now click on the **Animations** tab. Select the **toss_grenade** clip from the **Clips** list and leave the **Loop Pose** checkbox deselected. Under **Root Transform Rotation**, check **Bake into Pose** and set **Body Orientation** for **Baked Upon (at Start)**; Under **Root Transform Position (Y)**, check **Bake into Pose** and set **Original** for **Baked Upon (at Start)**; Under **Root Transform Position (XZ)**, leave the **Bake into Pose** checkbox deselected. Finally, click on the **Clamp Range** button to adjust the timeline. Click on **Apply** to confirm the changes.
7. Let's create our Mask. In the **Project** view, click on the **Create** button and add an Avatar Body Mask to the project. Name it `BodyMask`.

www.it-ebooks.info

8. Select the Body Mask and, in the **Inspector** view, deselect the character's legs, base, and IK spots, turning their outline red.



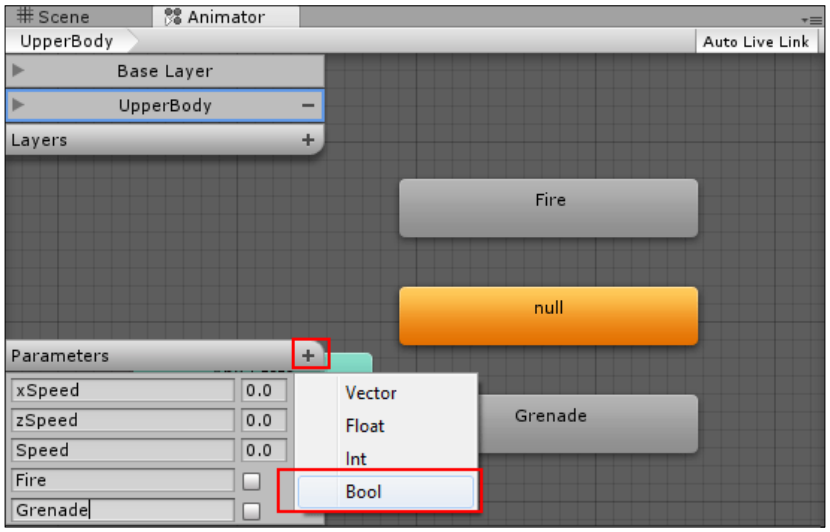
9. In the **Hierarchy** view, select the **Swat** model. Then, from the Animator component in the **Inspector** view, double-click on **swatController03** to open it.
10. In the **Animator** view, create a new Layer by clicking on the **+** sign on the **Layers** tab at the top-left corner of the screen, under **Base Layer**.
11. Name the new Layer `UpperBody` and select **BodyMask** for the **Human** field under **Masks**.



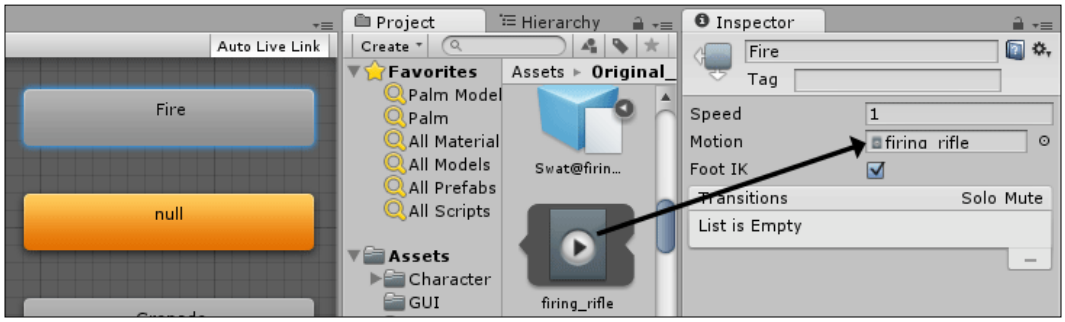
12. Now, in the **Animator** view, with the **UpperBody** layer selected, create three new empty states (by right-clicking on the gridded area and selecting **Empty** from the **Create State** menu). Name the default (orange colored) state `null`, and the other two, `Fire` and `Grenade`.

www.it-ebooks.info

13. In the **Parameters** field at the bottom-left corner of the screen, add two new parameters (of the type `Bool`), `Fire` and `Grenade`.



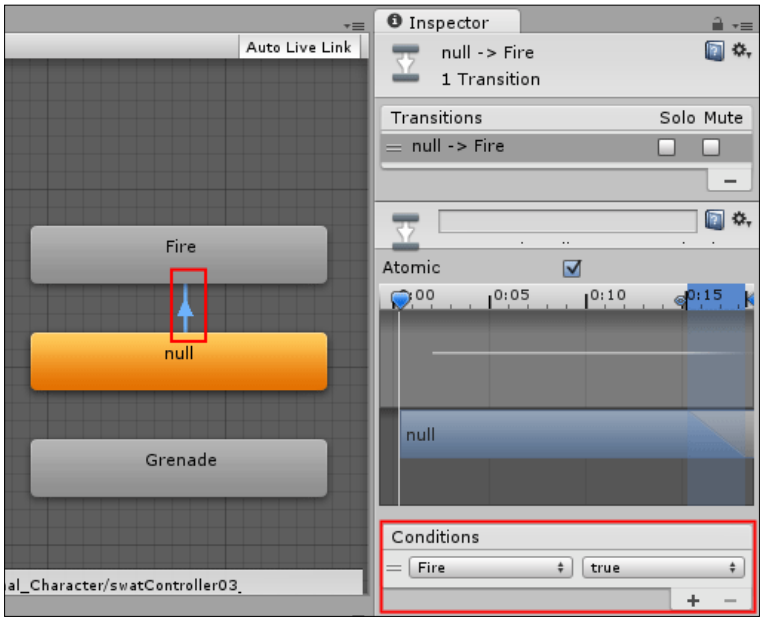
14. Select the **Fire** state and, in the **Inspector** view, add the **firing_rifle** animation clip to the **Motion** field.



15. Now select the **Grenade** state and, in the **Inspector** view, add the **toss_grenade** animation clip to the **Motion** field.
16. Right-click on the **null** state box and select **Make Transition** from the menu. Then, drag the white arrow into the box named **Fire**.

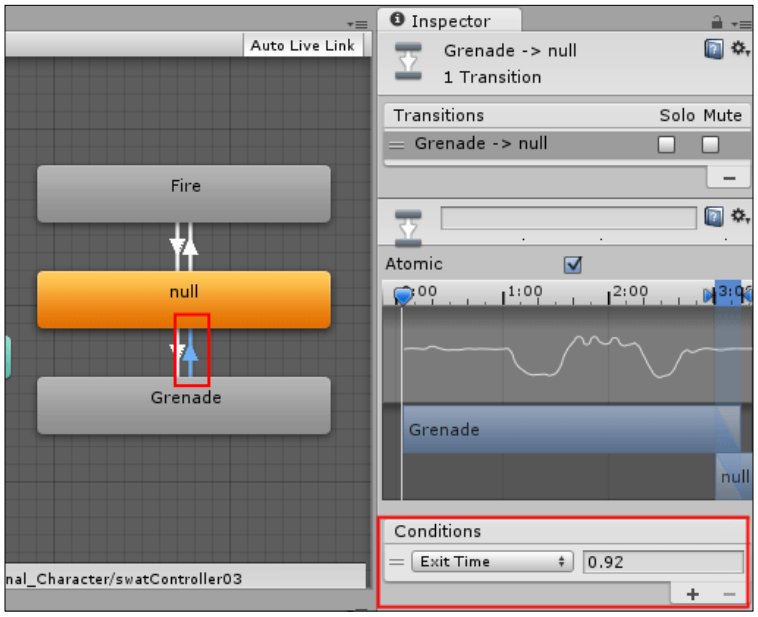
www.it-ebooks.info

17. Select the arrow (it should turn blue). Then, in the **Inspector** view, inside the **Conditions** box, select the options **Fire** and **true**.



18. Now make a transition from **null** to **Grenade**. Select the arrow and, in the **Conditions** box, select the options **Grenade** and **true**.
19. Now create transitions from **Fire** to **null**, and from **Grenade** to **null**. Select the arrow that goes from **Fire** to **null**, and in the **Conditions** box, select the options **Fire** and **false**.
20. Finally, select the arrow that goes from **Grenade** to **null**, and in the **Conditions** box, select the option **Exit Time** and enter the number `0.92`.

www.it-ebooks.info



21. In the **Hierarchy** view, select the **Swat** model. In the **Inspector** view, select the Basic Controller 03 component and open its script.
22. Immediately before the end of the **Update()** function, add the following code:

```
if(Input.GetKeyDown(KeyCode.F)){
  animator.SetBool("Grenade", true);
} else {
  animator.SetBool("Grenade", false);
}
if(Input.GetButtonDown("Fire1")){
  animator.SetBool("Fire", true);
}
if(Input.GetButtonUp("Fire1")){
  animator.SetBool("Fire", false);
}
```

23. Save the script and play your scene. You should be able to trigger the Fire and Grenade animations by clicking on the **Fire** button and pressing the *F* key, respectively. Observe how the character's legs respond to the walking and running animations.

www.it-ebooks.info

How it works...

Once the Avatar Body Mask has been created, it can be used as a way of filtering which body parts will actually play the animation states of a particular layer. In our case, we have constrained our **fire_rifle** and **toss_grenade** animation clips to the upper body of our character, leaving the lower body free to play the movement-related animation clips such as walking, running, and strafing.

There's more...

You might have noticed that the `UpperBody` layer has a parameter named `Blending`, which was set to `Override` by default. This means animation states in that layer will override animation states from lower layers when played. If changed to `Additive`, the animation from that layer would be added to the ones from lower layers.

For more information on Animation Layers and Avatar Body Masks, check out Unity's documentation at the following two locations:

`http://docs.unity3d.com/Documentation/Manual/AnimationLayers.html`

`http://docs.unity3d.com/Documentation/Manual/AvatarBodyMask.html`

Overriding Root Motion via script

Applying Root Motion to your character is a very practical and accurate way of animating it. However, every now and then you might need to manually control one or more aspects of the character's movement. Perhaps you only have an in-place animation to work with, or maybe you want the character's movement to be affected by other variables. In such cases, you will need to override Root Motion via script.

In this recipe, we will control the character's movement via script, making it rotate and jump.

Getting ready

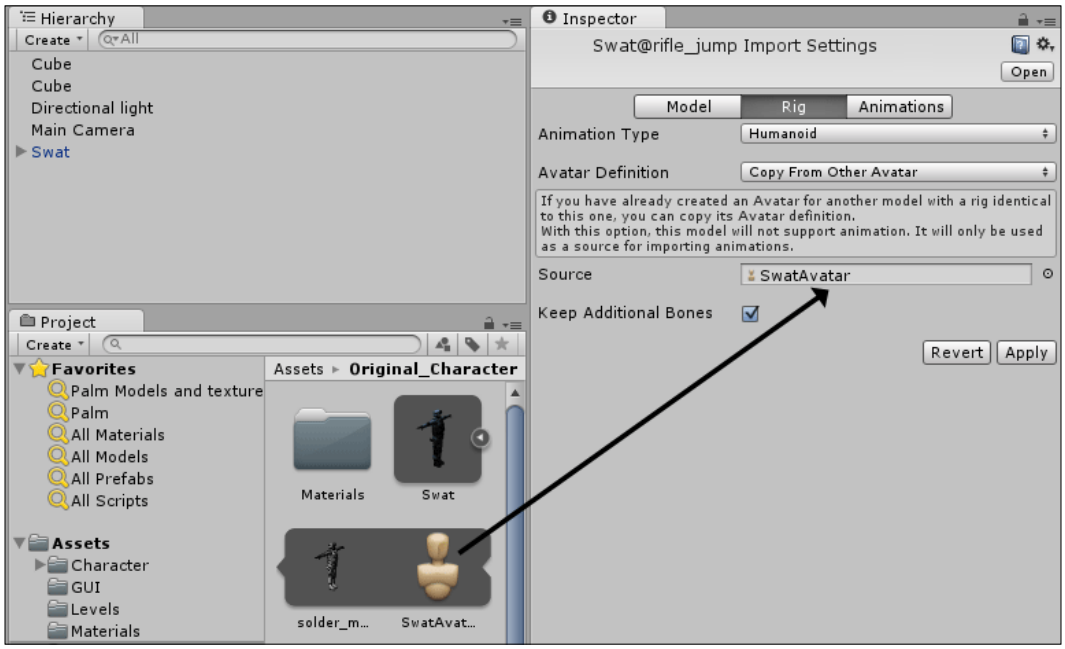
For this recipe, we have prepared a project named `MixamoProject`, containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

www.it-ebooks.info

How to do it...

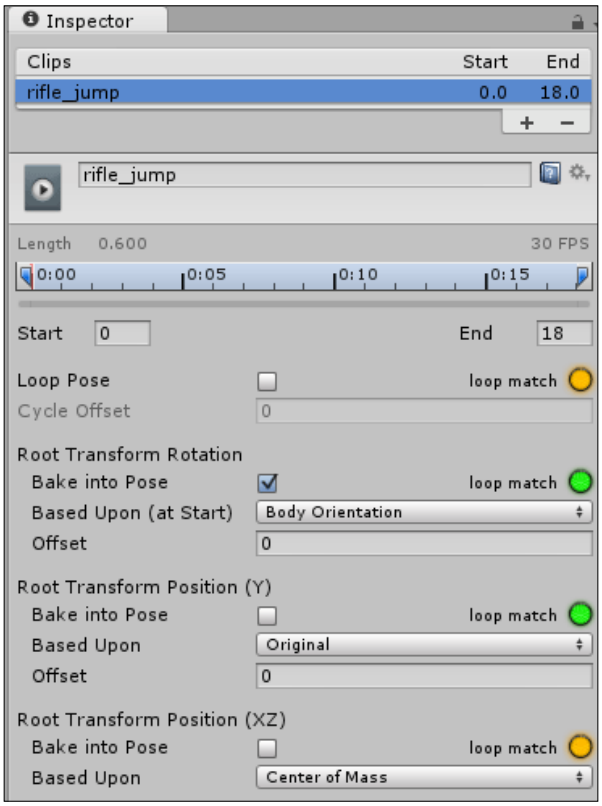
To apply Root Motion via script, perform the following steps:

1. Open the project, and then open the level named `05_04` (inside the `Levels` folder). Note that it includes a S.W.A.T. character, featuring the Animator component using the **swatController03** Controller.
2. We need to configure our animation clips. From the **Original_Character** folder, select the **Swat@rifle_jump** file.
3. Click on the **Rig** tab. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Copy From Other Avatar**. Then, select **swatAvatar** for **Source** by choosing it from the list or dragging it from the **Project** view into the slot. Confirm the changes by clicking on **Apply**.



www.it-ebooks.info

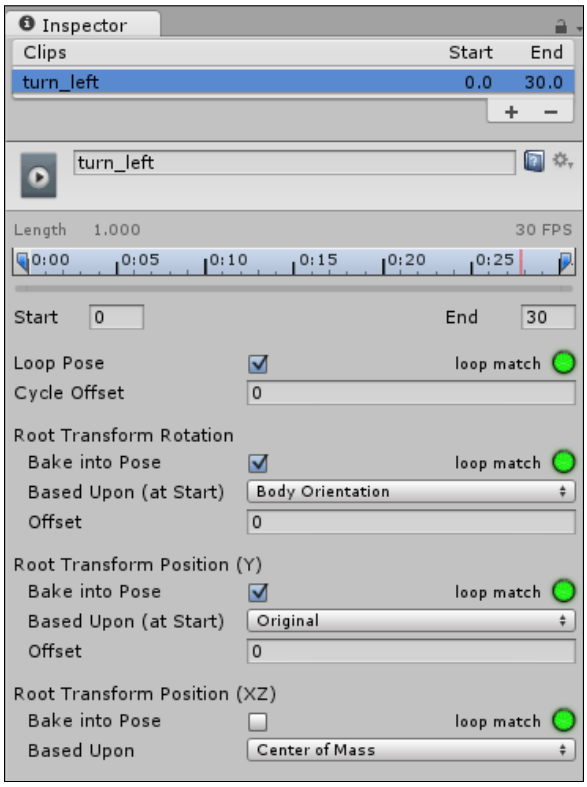
4. Now activate the **Animations** section. Select the clip **rifle_jump** from the **Clips** list and leave the Loop Pose checkbox deselected. **Under Root Transform Rotation**, check **Bake into Pose** and set **Body Orientation** for **Baked Upon (at Start)**; Under **Root Transform Position (Y)**, leave **Bake into Pose** deselected; Under **Root Transform Position (XZ)**, leave **Bake into Pose** deselected. Finally, click on the **Clamp Range** button to adjust the timeline. Click on **Apply** to confirm the changes.



5. Now, from the **Original_Character** folder, select the **Swat@turn_left** file.
6. Repeat step 3 of this recipe, applying **SwatAvatar** to **Swat@turn_left** as the **Source** for the **Humanoid Rig**.

www.it-ebooks.info

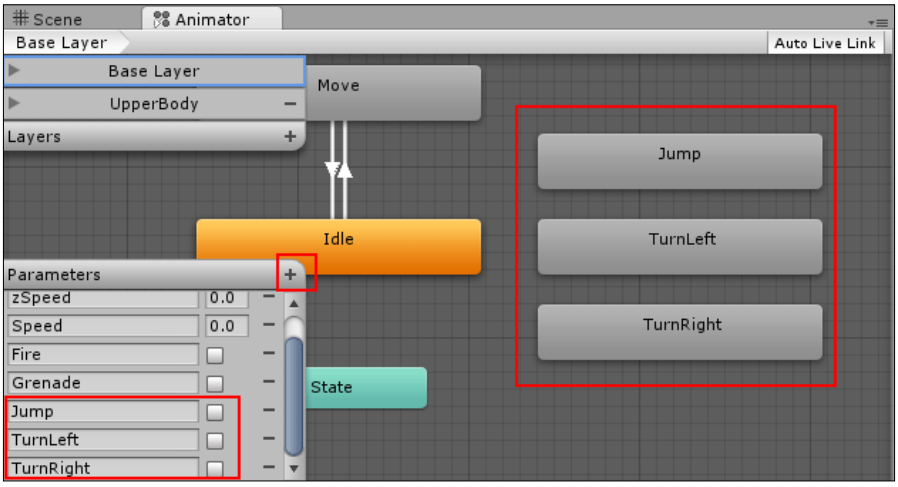
7. Now click on the **Animations** tab. Select the clip **turn_left** from the **Clips** list and select the **Loop Pose** checkbox. Under **Root Transform Rotation**, check **Bake into Pose** and set **Body Orientation** for **Baked Upon (at Start)**; Under **Root Transform Position (Y)**, check **Bake into Pose** and set **Original** for **Baked Upon (at Start)**; Under **Root Transform Position (XZ)**, leave **Bake into Pose** deselected. Finally, click on the **Clamp Range** button to adjust the timeline. Click on **Apply** to confirm the changes.



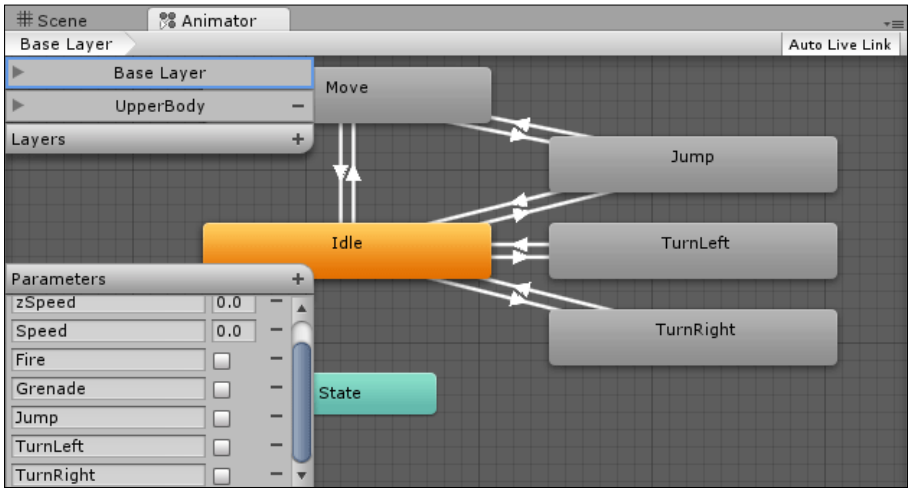
8. Repeat steps 6 and 7 of this recipe for the `Swat@turning_right_45_degrees` file.
9. From the **Hierarchy** view, select the **Swat** model. Then, from the Animator component in the **Inspector** view, open the **swatController04** controller.

www.it-ebooks.info

10. Using the **Parameters** widget, create the Boolean parameters `Jump`, `TurnLeft`, and `TurnRight`. Then, in the gridded area, create the states **Jump**, **TurnLeft**, and **TurnRight**.

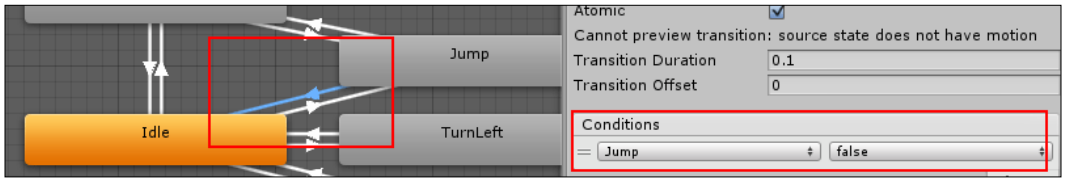


11. By right-clicking on each state and selecting the **Make Transition** option, create transitions between **Jump** and **Idle**, **Jump** and **Move**, **Turn Left** and **Idle**, and **Turn Right** and **Idle**. You should add return transitions as well.



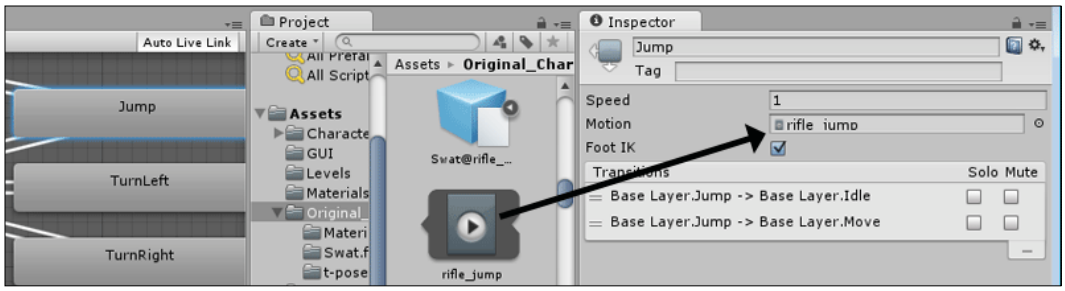
12. Select the arrow that goes from **Idle** to **Jump**. It should turn blue in color. Then, in the **Inspector** view, in the **Conditions** box, select the options **Jump** and **true**. Now select the arrow that goes back from **Jump** to **Idle**, and select **Jump** and **false** in its **Conditions** box.

www.it-ebooks.info



13. Repeat the previous step with the arrow that goes from **Move** to **Jump** and vice-versa.
14. Again, apply the instructions from step 12 onwards to the arrows that go from **TurnLeft** and **TurnRight** to **Idle** and vice-versa. In the **Conditions** box, use **TurnLeft** and **TurnRight** (instead of **Jump**) according to the selected state.
15. Select the **Jump** state and, in the **Inspector** view, apply the animated clip from **Swat:rifle_jump** to its **Motion** field.

You can experiment with widening the blue segment of the timeline illustrating the transition between **Jump** and **Idle** (and vice-versa), using the handles to get a smoother transition.



16. As described in the previous step, apply the animation clips from **Swat:turn_left** and **Swat:turning_right_45_degrees** to the Motion clips of **TurnLeft** and **TurnRight** respectively.
17. From the **Hierarchy** view, select the **Swat** model. Then, in the **Inspector** view, open the script from the Basic Controller 04 component.
18. Immediately after the line `public float transitionTime = 0.25f;`, add the following code:

```
public float jumpSpeed = 4.0F;
public float gravity = 20.0F;
private float jumpPos = 0.0f;
private float verticalSpeed = 0;
private float xVelocity = 0.0f;
private float zVelocity = 0.0f;
```


www.it-ebooks.info

19. Immediately after the line `if(controller.isGrounded){`, add the following code:

```
if (Input.GetKey(KeyCode.Space)) {
    animator.SetBool("Jump", true);
    verticalSpeed = jumpSpeed;
}else{
    animator.SetBool("Jump", false);
}
if(Input.GetKey(KeyCode.Q)){
    animator.SetBool("TurnLeft", true);
    transform.Rotate(Vector3.up * (Time.deltaTime * -45.0f),
Space.World);
} else {
    animator.SetBool("TurnLeft", false);
}
if(Input.GetKey(KeyCode.E)){
    animator.SetBool("TurnRight", true);
    transform.Rotate(Vector3.up * (Time.deltaTime * 45.0f), Space.
World);
} else {
    animator.SetBool("TurnRight", false);
}
```

20. Finally, add the following lines immediately before the final closing bracket of the code:

```
    void OnAnimatorMove(){
        Vector3 deltaPosition = animator.deltaPosition;
        if(controller.isGrounded){
           xVelocity = animator.GetFloat("Speed")  * controller.
           velocity.x * 0.25f;
           zVelocity = animator.GetFloat("Speed") * controller.
           velocity.z * 0.25f;
        }
        verticalSpeed += Physics.gravity.y * Time.deltaTime;
        if(verticalSpeed <= 0){
            animator.SetBool("Jump", false);
        }
        deltaPosition.y = verticalSpeed * Time.deltaTime;
        if(!controller.isGrounded){
          deltaPosition.x = xVelocity * Time.deltaTime;
          deltaPosition.z = zVelocity * Time.deltaTime;
        }
        controller.Move(deltaPosition);
```


www.it-ebooks.info

```
            if ((controller.collisionFlags & CollisionFlags.Below) !=
0){
                verticalSpeed = 0;
            }
            transform.rotation = animator.rootRotation;
        }
```

21. Save your script and play the scene. You should be able to jump and turn around by using the Space bar, *Q*, and *E* keys respectively. Observe how the character can also turn around while moving.

How it works...

We took two different paths to modify our character's transform settings. First, we used the `Rotate` command, as in `transform.Rotate(Vector3.up * (Time.deltaTime * -45.0f), Space.World)`, to make the character actually turn around when the *Q* and *E* keys are held down. This command was used in conjunction with `animator.SetBool("TurnLeft", true)`, which triggered the correct animation clip.

Also, we used Unity's `OnAnimatorMove()` function for overriding the animation's original Root Motion. Observe that, once this function is added to the script, the field **Apply Root Motion** on the Animator component changes from a checked box to **Handled by Script**. In our case, we used it to learn about the character's speed and direction while he is grounded, in order to apply it once he jumps. We also apply gravity when calculating his position, in case his feet are not touching any surface.

There's more...

Want to change the character's direction using a horizontal mouse movement? Just uncomment the line `transform.Rotate(Vector3.up * (Time.deltaTime * v * Input.GetAxis("Mouse X") * 90), Space.World)`.

Adding rigid props to animated characters

In case you haven't included a sufficient number of props for your character when modeling and animating it, you might want to give him the chance to collect new ones at runtime. In this recipe, we will learn how to instantiate a game object and assign it to a character, respecting the animation hierarchy.

Getting ready

For this recipe, we have prepared a project named `MixamoProject`, containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

www.it-ebooks.info

How to do it...

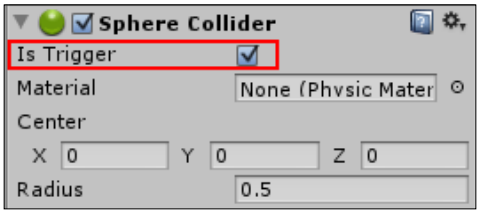
To add a rigid prop to an animated character at runtime, perform the following steps:

1. Open the **MixamoProject** project, and then open the level named **05_05** in the **Levels** folder.
2. You should see Mixamo's animated S.W.A.T. soldier and two spheres. Those spheres will trigger the addition of new props for the character.
3. In the **Project** view, create a new C# script named `AddProp.cs`.
4. Open the script and add the following code:

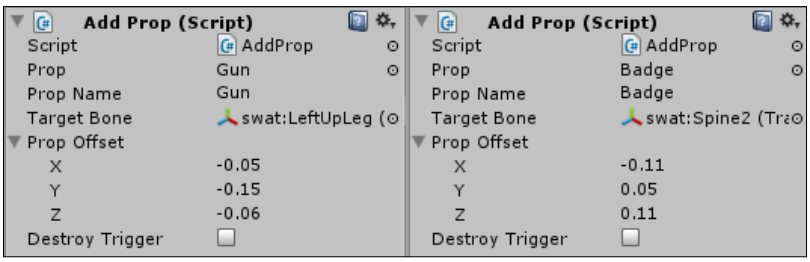
```
using UnityEngine;
using System.Collections;

public class AddProp : MonoBehaviour {
    public GameObject prop;
    public string propName;
    public Transform targetBone;
    public Vector3 propOffset;
    public bool destroyTrigger = true;

    void OnTriggerEnter ( Collider collision){

          if (targetBone.IsChildOf(collision.transform)){
           bool checkProp = false;
           foreach(Transform child in targetBone){
               if (child.name == propName)
                    checkProp = true;
           }

          if(!checkProp){
           GameObject newprop;
       newprop = Instantiate(prop, targetBone.position,
targetBone.rotation) as GameObject;
           newprop.name = propName;
           newprop.transform.parent = targetBone;
           newprop.transform.localPosition += propOffset;
           if(destroyTrigger)
               Destroy(gameObject);
              }
          }
    }
}
```


www.it-ebooks.info

5. Save and close the script.
6. Attach the **AddProp.cs** script by dragging it from the **Project** view onto the objects named **Sphere Blue** and **Sphere Red** in the **Hierarchy** view.
7. Select each sphere, and select the **Is Trigger** checkbox in the **Sphere Collider** component (located in the **Inspector** view).



8. Select **Sphere Blue** and check out its **Add Prop** component.
9. First, let's assign a Prefab to the **Prop** field. In the **Project** view, expand the **Props** folder. Then, drag the Prefab named **Gun** into the **Prop** field.
10. In the **Prop Name** field, type in **Gun**.
11. In the **Target Bone** field, select the **swat:LeftUpLeg** transform from the list (or drag it from the **Hierarchy** view; it is a child of the **Character** game object).
12. Expand the **Prop Offset** field and type in the values of **X**, **Y**, and **Z** as `-0.05`, `-0.15`, and `-0.06` respectively. Finally, deselect the **Destroy Trigger** checkbox.
13. Select the **Sphere Red** object and fill in the variables as follows: **Prop**: `Badge`; **Prop Name**: `Badge`; **Target Bone**: `swat:Spine2`; **Prop Offset**: **X**: `-0.11`, **Y**: `0.05`, **Z**: `0.11`. Also, deselect the **Destroy Trigger** checkbox.



www.it-ebooks.info

14. Play the scene. Use the WASD keyboard control scheme and direct the character to the spheres. Colliding with them will add a new handgun holster (to be attached to the character's left leg) and a badge (to be attached on the left-hand side of the character's chest).

How it works...

Once it's been triggered by the character, the script attached to the spheres instantiates the assigned Prefabs, making them children of the bones they have been placed into. The **Prop Offset** field can be used to fine-tune the exact position of the prop (relative to its parent transform). As the props become parented by the bones of the animated character, they will follow and respect its hierarchy and animation. Note that the script checks for pre-existing props of the same name before actually instantiating new ones.

There's more...

You could make a similar script to remove props. In that case, the function `OnTriggerEnter` would contain only the following code:

```
if (targetBone.IsChildOf(collision.transform)){
    foreach(Transform child in targetBone){
        if (child.name == propName)
            Destroy (child.gameObject);
    }
}
```


www.it-ebooks.info

Making an animated character throw an object

Now that your animated character is ready, you might want to coordinate his actions with his animation states. In this recipe, we will exemplify this by making the character throw an object when the player presses the appropriate button. Also, we will make sure that the action corresponds to the character's animation.

Getting ready

For this recipe, we have prepared a project named `MixamoProject`, containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

How to do it...

To make an animated character throw an Easter egg(!), follow these steps:

1. Open the **MixamoProject** project, and then open the level named **05_06** (in the **Levels** folder).
2. Play the level and press the *F* key on your keyboard. The character will move as if he's throwing something with his right hand.
3. In the **Project** view, create a new C# script named `ThrowObject.cs`.
4. Open the script and add the following code:

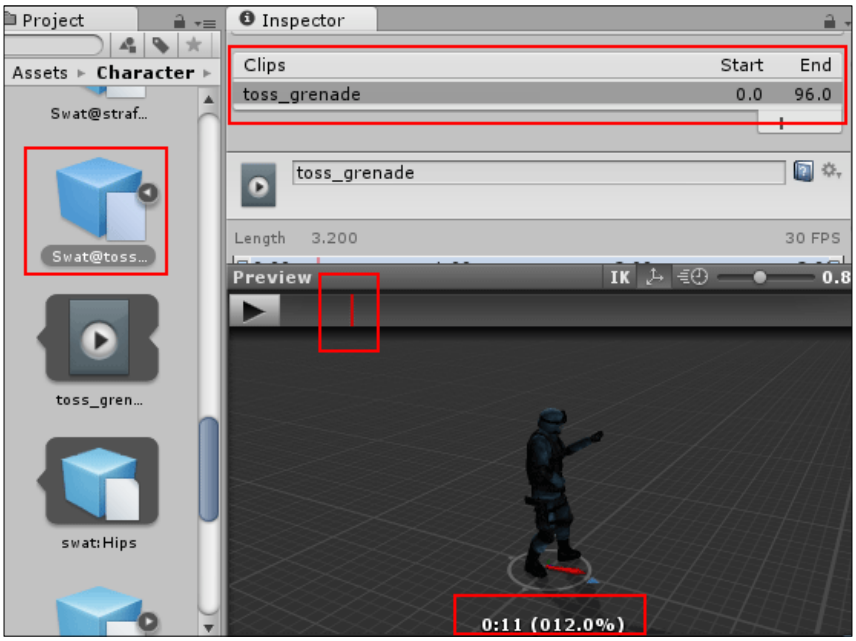
```
using UnityEngine;
using System.Collections;

public class ThrowObject : MonoBehaviour {
    public GameObject projectile;
    public Vector3 projectileOffset;
    public Vector3 projectileForce;
    public Transform charactersHand;
    public float lenghtPrepare;
    public float lenghtThrow;
    public float compensationYAngle = 20.0f;
    private bool prepared = false;
    private bool threw = false;
    private Animator animator;
    public void Start(){
        animator = GetComponent<Animator>();
    }
    public void LateUpdate(){
```


www.it-ebooks.info

```
        AnimatorStateInfo stateInfo = animator.
GetCurrentAnimatorStateInfo(1);

        if(stateInfo.IsName("UpperBody.Grenade")){

            if(stateInfo.normalizedTime >= lenghtPrepare * 0.01 &&
!prepared)
                Prepare();

            if(stateInfo.normalizedTime >= lenghtThrow * 0.01 &&
!threw)
                Throw();

        } else {
        prepared = false;
        threw = false;
    }
}

    public void Prepare () {
        prepared = true;
        projectile = Instantiate(projectile, charactersHand.
position, charactersHand.rotation) as GameObject;
        if(projectile.GetComponent<Rigidbody>())
            Destroy(projectile.rigidbody);

        projectile.GetComponent<SphereCollider>().enabled = false;
        projectile.name = "projectile";
        projectile.transform.parent = charactersHand;
        projectile.transform.localPosition = projectileOffset;
        projectile.transform.localEulerAngles = Vector3.zero;
    }
    public void Throw () {
        threw = true;
        Vector3 dir = transform.rotation.eulerAngles;
        dir.y += compensationYAngle;
        projectile.transform.rotation = Quaternion.Euler(dir);
        projectile.transform.parent = null;
        projectile.GetComponent<SphereCollider>().enabled = true;
        projectile.AddComponent<Rigidbody>();
        Physics.IgnoreCollision(projectile.collider, collider);
        projectile.rigidbody.AddRelativeForce(projectileForce);
    }
}
```

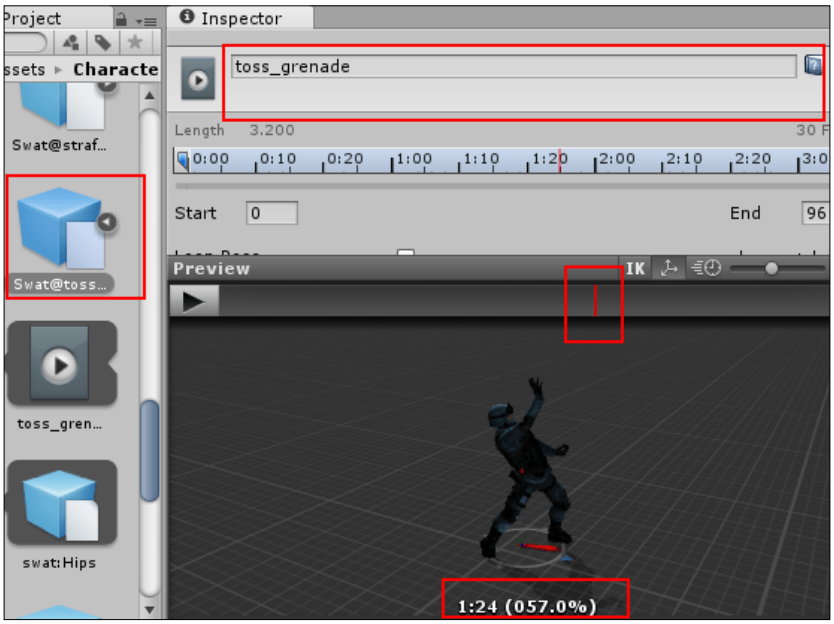

www.it-ebooks.info

5. Save and close the script.
6. Attach the **ThrowObject.cs** script to the game object named **Swat**.
7. Select the **Swat** object. In the **Inspector** view, check out its **Throw Object** component. From the **Project** view, drag the Prefab named **EasterEgg** (available in the **Props** folder) into the **Projectile** field.
8. In the **Characters Hand** field, select the **swat:RightHand** transform.
9. As we are still on the **Throw Object** component, fill in the variables as follows: **Projectile Offset**: **X**: `-0.07`, **Y**: `-0.04`, **Z**: `0`; **Projectile Force**: **X**: `0`, **Y**: `200`, **Z**: `500`.
10. We need to find out the values that should go into the **Length Prepare** and **Length Throw** fields. In the **Project** view, select **Swat@toss_grenade**.
11. In the **Inspector** view, select the clip **toss_grenade** and drag the playhead through the timeline. Observe how the character starts preparing for the throw by grabbing the grenade at 11 seconds (12 percent of the length). This is shown in the following screenshot:

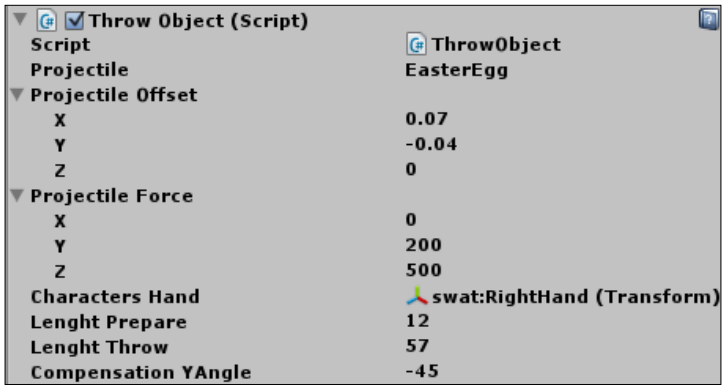


www.it-ebooks.info

Likewise, it finishes the throw movement around 1:24 (57 percent of the length), as shown in the following screenshot:



12. Again, in the **Hierarchy** view, select the **Swat** game object.
13. In the **Inspector** view, change the values in the **Length Prepare** and **Length Throw** fields to `12` and `57` respectively.
14. Finally, change the value of **Compensation YAngle** to `-45`.



15. Play your scene. Your character should now be able to throw an Easter egg when you press the *F* key.

www.it-ebooks.info

How it works...

Once the **toss_grenade** animation reaches 12 percent of its length, our script detects it and calls the function named `Prepare()`.This function instantiates a Prefab, now named projectile, into the character's hand. The values specified under **Projectile Offset** are used to fine-tune its position, making it respect the character's hierarchy. Also, it disables the Prefab's collider and destroys its Rigidbody component, provided it has one.

Later, when the **toss_grenade** animation reaches 57 percent of its length, the `Throw()` function is called, which enables the projectile's collider, adds a Rigidbody component to it, and makes it independent of the Character object. Finally, it adds a relative force to the projectile's Rigidbody component so it behaves as if it was thrown by the character. **Compensation YAngle** is used to adjust the direction of the grenade.

Applying ragdoll physics to a character

Action games often make use of ragdoll physics to simulate the character's body reaction to being unconsciously under the effect of a hit or explosion. In this recipe, we will learn how to set up and activate ragdoll physics for our character whenever he steps on a landmine object. We will also use this opportunity to reset the character's position and animations a number of seconds after that event.

Getting ready

For this recipe, we have prepared a project named `MixamoProject` containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

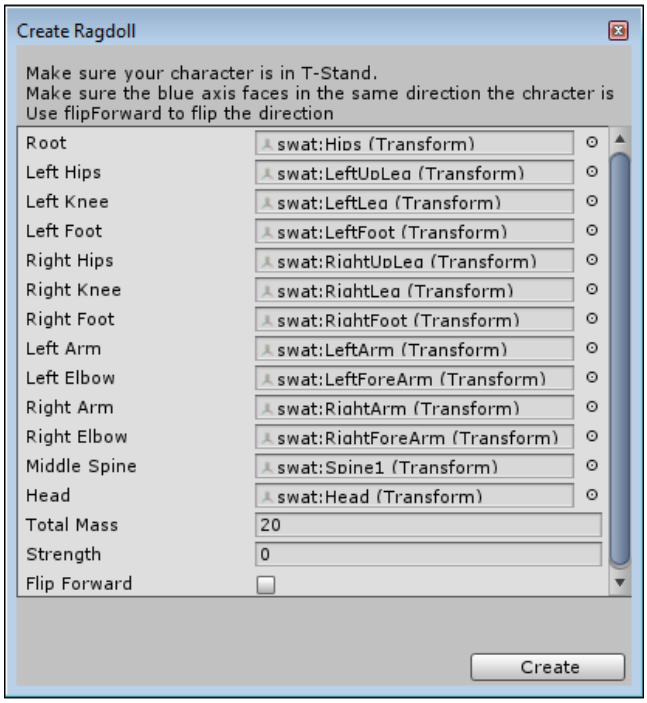
How to do it...

To apply ragdoll physics to your character, follow these steps:

1. Open the **MixamoProject** project, and then open the level named **05_07** (in the **Levels** folder).
2. You should see Mixamo's animated S.W.A.T. soldier and two cylinders (the landmine and the spawn point).
3. First, let's set up our ragdoll. Go to **GameObject** | **Create Other** | **Ragdoll...**. The Ragdoll Wizard should pop up.
4. Assign the transforms as follows (these are also shown in the following screenshot):
    - **Root**: **swat:Hips**
    - **Left Hips**: **swat:LeftUpLeg**

www.it-ebooks.info

- **Left Knee**: **swat:LeftLeg**
- **Left Foot**: **swat:LeftFoot**
- **Right Hips**: **swat:RightUpLeg**
- **Right Knee**: **swat:RightLeg**
- **Right Foot**: **swat:RightFoot**
- **Left Arm**: **swat:LeftArm**
- **Left Elbow**: **swat:LeftForeArm**
- **Right Arm**: **swat:RightArm**
- **Right Elbow**: **swat:RightForeArm**
- **Middle Spine**: **swat:Spine1**
- **Head**: **swat:Head**



5. In the **Project** view, create a new C# script named `RagdollCharacter.cs`.
6. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;
```


www.it-ebooks.info

```
public class RagdollCharacter : MonoBehaviour {
    private float hitTime;
    private bool wasHit = false;

    void Start () {
        DeactivateRagdoll();
    }
    void Update () {
        if(wasHit){
            if(Time.time >= hitTime + 5.0f)
                DeactivateRagdoll();
            }
        }
    public void ActivateRagdoll(){
        this.GetComponent<BasicController>().enabled = false;
        this.GetComponent<Animator>().enabled = false;
        foreach(Rigidbody bone in GetComponentsInChildren<Rigidbo
dy>()){
            bone.isKinematic = false;
            bone.detectCollisions = true;
        }
        wasHit = true;
        hitTime = Time.time;
    }
    public void DeactivateRagdoll(){
        this.GetComponent<BasicController>().enabled = true;
        this.GetComponent<Animator>().enabled = true;

        foreach(Rigidbody bone in GetComponentsInChildren<Rigidbo
dy>()){
            bone.isKinematic = true;
            bone.detectCollisions = false;
        }
        transform.position = GameObject.Find("Spawnpoint").
transform.position;
        transform.rotation = GameObject.Find("Spawnpoint").
transform.rotation;
        wasHit = false;
    }
}
```

7. Save and close the script.
8. Attach the **RagdollCharacter.cs** script to the **Character** game object.
9. In the **Project** view, create a new C# script named `Landmine.cs`.

www.it-ebooks.info

10. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;

public class Landmine : MonoBehaviour {
    public float range = 50.0f;
    public float force = 2000.0f;

    void OnTriggerEnter ( Collider collision ){
        if(collision.gameObject.tag == "Player"){
collision.GetComponent<RagdollCharacter>().ActivateRagdoll();
            Vector3 explosionPos = transform.position;
            Collider[] colliders = Physics.
OverlapSphere(explosionPos, range);
            foreach (Collider hit in colliders) {
                if (hit.rigidbody)
                    hit.rigidbody.AddExplosionForce(force,
explosionPos, range, 3.0F);
            }
        }
    }
}
```

11. Save and close the script.
12. Attach the script to the **Landmine** game object.
13. Play the scene. Use the WASD keyboard control scheme to direct the character to the **Landmine** game object. Colliding with it will activate the character's ragdoll physics and apply an explosive force to it. As a result, the character will be thrown to a considerable distance.

How it works...

Unity's Ragdoll Wizard assigns the Collider, Rigidbody, and Character Joint components to selected transforms. When used in conjunction with each other, these components make ragdoll physics possible. However, they must be disabled whenever we want our character to be animated or controlled by the player. In our case, we switch those components on and off using the `RagdollCharacter` script and its two functions, `ActivateRagdoll()` and `DeactivateRagdoll()` the latter includes instructions to re-spawn our character in the appropriate place.

For testing purposes, we have also created the Landmine script, which calls RagdollCharacter's function, `ActivateRagdoll()`. It also applies an explosive force to our ragdoll character, violently throwing him outside the explosion site.

www.it-ebooks.info

There's more...

Instead of resetting the character's transform settings, you could have destroyed his game object and instantiated a new one over the re-spawn point using tags. For more information on that subject, check Unity's documentation at the following location: `http://docs.unity3d.com/Documentation/ScriptReference/GameObject.FindGameObjectsWithTag.html`

Rotating the character's torso to aim

When you're playing a third person character, you might want your character to aim his weapon at a target that is not directly in front of him, without making him change his direction. In those cases, you will need to apply what is known as *procedural animation*; it does not rely on pre-made animation clips, but rather on the processing of other data such as player input, to animate the character. In this recipe, we will use this technique to rotate the character's torso.

Getting ready

For this recipe, we have prepared a project named `MixamoProject`, containing several assets such as levels, animated characters, and props. You can find it inside the `0423_05_codes` folder.

How to do it...

1. Open the **MixamoProject** project, and then open the level named **05_08** (under the **Levels** folder).
2. You should see Mixamo's animated S.W.A.T. soldier. We have included three different cameras for you to experiment with (they are children of the Character game object).
3. In the **Project** view, create a new C# script named `MouseAim.cs`.
4. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;

public class MouseAim : MonoBehaviour {

    public Transform spine;
    public Transform armedHand;
    public bool lockY = false;
    public float compensationYAngle = 20.0f;
```

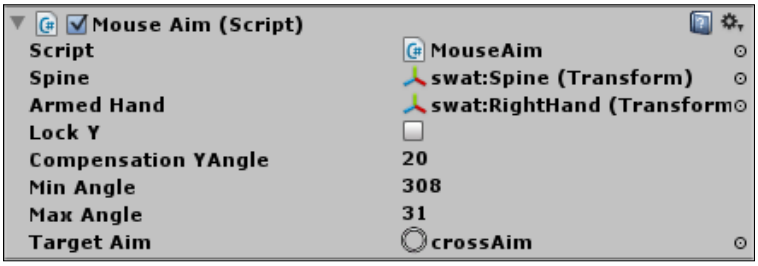

www.it-ebooks.info

```
    public float minAngle = 308.0f;
    public float maxAngle = 31.0f;
    public Texture2D targetAim;
    private Vector2 aimLoc;
    private bool onTarget = false;

    public void LateUpdate(){
        Vector3 point = Camera.main.ScreenToWorldPoint(new
Vector3(Input.mousePosition.x, Input.mousePosition.y, Camera.main.
farClipPlane));
        if(lockY)
            point.y = spine.position.y;
        Vector3 relativePoint = transform.
InverseTransformPoint(point.x, point.y, point.z);
        if(relativePoint.z < 0){
            Vector3 inverseZ = transform.InverseTransformPoint(rel
ativePoint.x,relativePoint.y,-relativePoint.z);
            point = inverseZ;
        }
        spine.LookAt(point, Vector3.up);
        Vector3 comp = spine.localEulerAngles;
        comp.y = spine.localEulerAngles.y + compensationYAngle;
        spine.localEulerAngles = comp;
        if(spine.localEulerAngles.y > maxAngle && spine.
localEulerAngles.y < minAngle){
            if(Mathf.Abs((spine.localEulerAngles.y - minAngle)) <
Mathf.Abs((spine.localEulerAngles.y - maxAngle))){
                Vector3 min = spine.localEulerAngles;
                min.y = minAngle;
                spine.localEulerAngles = min;
            } else {
                Vector3 max = spine.localEulerAngles;
                max.y = maxAngle;
                spine.localEulerAngles = max;
            }
        }
        RaycastHit hit;
        if (Physics.Raycast (armedHand.position, point, out hit))
{
            onTarget = true;
            aimLoc =  Camera.main.WorldToViewportPoint(hit.point);
```


www.it-ebooks.info

```
            } else {
                onTarget = false;
                aimLoc = Camera.main.WorldToViewportPoint(point);
            }
            //Debug.DrawRay (armedHand.position, point, Color.red);
        }

        void OnGUI(){
            int sw = Screen.width;
            int sh = Screen.height;
            GUI.DrawTexture(new Rect(aimLoc.x * sw - 8, sh-(aimLoc.y *
sh) -8, 16, 16), targetAim, ScaleMode.StretchToFill, true, 10.0F);
        }
}
```

5. Save and close the script.
6. Attach the **MouseAim.cs** script by dragging it from the **Project** view into the **Swat** game object.
7. Select the character, and in the **Mouse Aim** component (located in the **Inspector** view), fill in or assign the variables as follows: **Spine**: **swat:Spine**; **Armed Hand**: **swat:RightHand**; **Lock Y**: leave it deselected (unless you're using **Camera_top**); **Compensation Y Angle**: `20`; **Min Angle**: `308`; **Max Angle**: `31`. Also, drag the **crossAim** texture from the **GUI** folder in the **Project** view, into the **Target Aim** field.



www.it-ebooks.info

8. Play the scene. You should now be able to rotate the character's torso by moving the mouse cursor around the screen. Even better, a GUI texture will be displayed wherever he is aiming at.

How it works...

Our script starts by converting our bi-dimensional mouse cursor screen's coordinates to three-dimensional world space coordinates (stored in the `point` variable). Then, it rotates the character's torso towards the point's location using the `LookAt()` command. Additionally, it makes sure the spine does not extrapolate **Min Angle** and **Max Angle**, which could cause distortions to the character model.

In the process, we convert the target point from its absolute world coordinates to its position in relation to the character, and vice-versa. This is done so that we can detect whether the target point is located behind the character, and if it is, manipulate the point so that the character keeps aiming forwards.

Also, we have included a `Compensation YAngle` variable that makes it possible for us to fine-tune the character's alignment with the mouse cursor.

Please note that all the commands are inside the `LateUpdate()` method. This is done to make sure that our transform manipulations override the character's animation clips.

www.it-ebooks.info

There's more...

In case you want to implement shooting, we have provided you with a place to start by casting a ray from the character's armed hand to the target point. To see it as you test the scene, uncomment the line `Debug.DrawRay (armedHand.position, point, Color.red);`.

The ray cast detects collisions with surrounding objects for which the **targetAim** texture map, usually drawn over the mouse cursor location, should snap.

www.it-ebooks.info

www.it-ebooks.info

6
Playing and Manipulating Sounds

In this chapter, we will cover:

- Matching audio pitch to animation speed
- Adding customizable volume controls
- Simulating a tunnel environment with Reverb Zones
- Preventing the AudioClip from restarting if already playing
- Waiting for audio to finish before auto-destructing an object
- Making a dynamic soundtrack

Introduction

Sound is a very important part of the gaming experience. In fact, we can't stress enough how crucial it is to the player's immersion in a virtual environment. Just think of the engine running in your favorite racing game, the distant urban buzz in a simulator game, or the creeping noises in horror games. Think of how those sounds transport you into the game. This chapter is filled with recipes that, hopefully, will help you implement a better and more efficient sound design to your projects.

Matching audio pitch to animation speed

Many artifacts sound higher in pitch when accelerated and lower when slowed down. Car engines, fan coolers, a Vinyl record player... the list goes on. If you want to simulate this kind of sound effect in an animated object that can have its speed changed dynamically, follow this recipe.

www.it-ebooks.info

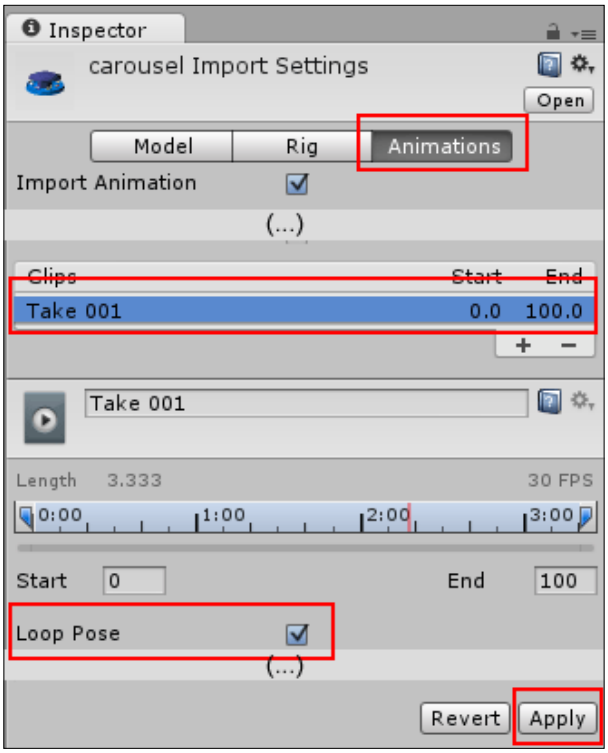
Getting ready

For this recipe, you'll need an animated 3D object and an audio clip. Please use the `carousel.fbx` and `carouselSound.wav` files, available in the `0423_06_01` folder.

How to do it...

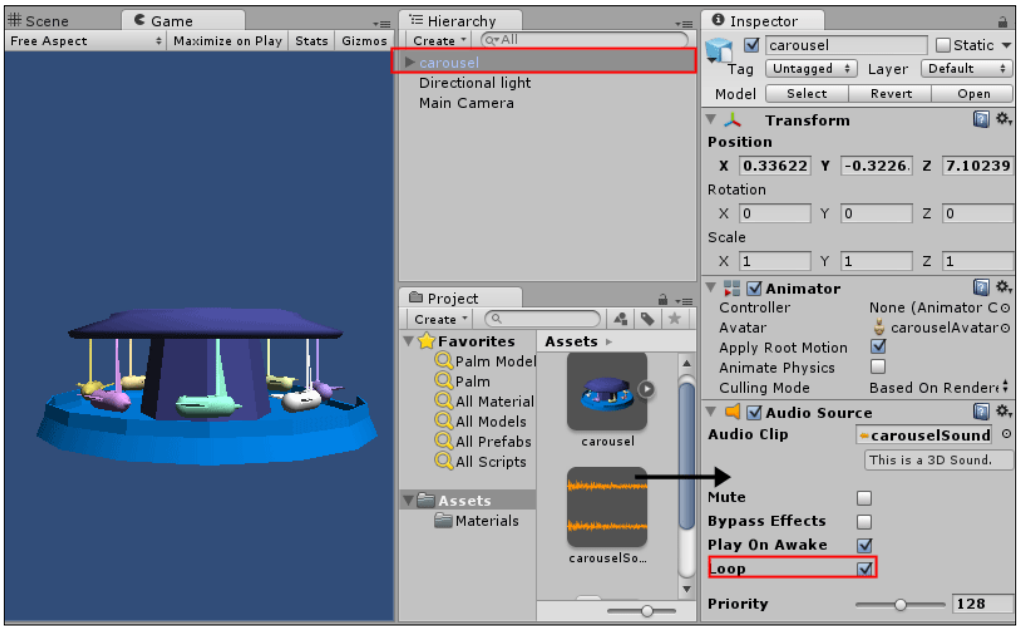
To change the pitch of an audio clip according to the speed of an animated object, please follow these steps:

1. Import the `carousel.fbx` file into your Unity project.
2. Select the `carousel.fbx` file in the **Project** view. Then, in the **Inspector** view, check its **Import Settings**. Under **Animations**, select the **Take 001** clip and make sure to check the **Loop Pose** option. Click the **Apply** button to save changes:



www.it-ebooks.info

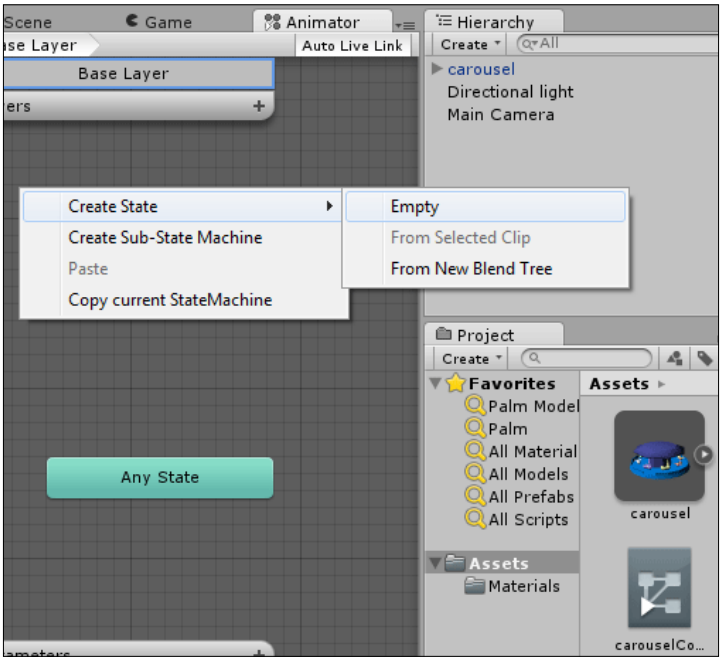
3. Add the carousel to the scene by dragging it from the **Project** view into the **Hierarchy** view.
4. Add a **Directional Light** to the scene through the **Create** drop-down menu on top of the **Hierarchy** view.
5. Import the `carouselSound.wav` audio clip file.
6. Select the **carousel** game object and drag **carouselSound** from the **Project** view into the **Inspector** view, adding it as an audio source for that object.
7. In the **Audio Source** component of the carousel, check the box for the **Loop** option:



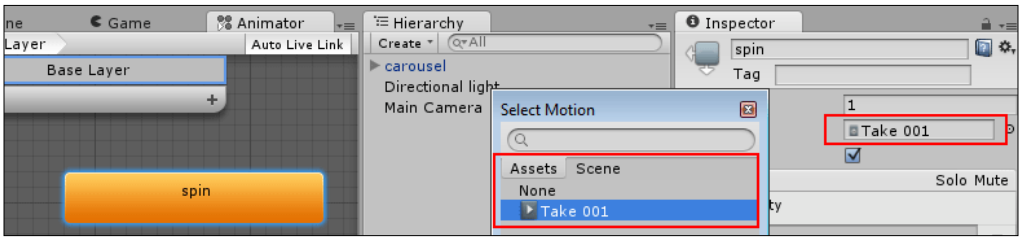
8. We need to create a controller for our object. In the **Project** view, click the **Create** button and select **Animator Controller**. Name it `CarouselController`.

www.it-ebooks.info

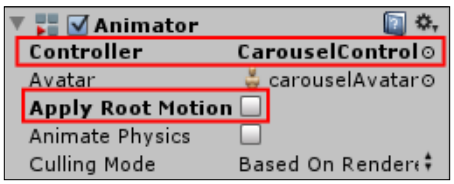
9. Double click **CarouselController** to open the **Animator** view. Then, right-click the gridded area and select **Create State** | **Empty** from the contextual menu:



10. Name the new state **spin** and set **Take 001** as its motion in the **Motion** field:



11. From the **Hierarchy** view, select the carousel. Then, in the **Animator** component (in the **Inspector** view), set **CarouselAnimator** as its **Controller** and uncheck the **Apply Root Motion** option:



www.it-ebooks.info

12. In the **Project** window, create a new C# script and rename it as `ChangePitch`.
13. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;

public class ChangePitch : MonoBehaviour{
   public float speed = 0.0f;
   public float minSpeed = 0.0f;
   public float maxSpeed = 2.0f;
   public float animationSoundRatio = 1.0f;
   private Animator animator;
 void Start(){
     animator =  GetComponent<Animator>();
 }
    void Update(){
        animator.speed = speed;
        audio.pitch = speed * animationSoundRatio;
    }
    void OnGUI(){
        Rect rect = new Rect(10, 10, 100, 30);
        speed = GUI.HorizontalSlider(rect, speed, minSpeed,
maxSpeed);
    }
}
```

14. Save your script and add it as a component to the carousel.
15. Play the scene and change the animation speed, along with the audio pitch, using the slidebar.

How it works...

The idea behind the script and its implementation are actually quite straightforward. It creates a slidebar from which the user can change the speed of the animator component. Then, it updates the audio pitch based on that number.

There's more...

Here is some information on how to fine-tune and customize this recipe.

Changing the Animation / Sound Ratio parameter

If you want the audio clip pitch to be either more or less affected by the animation speed, change the value of the **Animation / Sound Ratio** parameter.

www.it-ebooks.info

Adding customizable volume controls

Sound volume adjustment can be a very important feature, especially if your game is a standalone. After all, it can be very frustrating needing to access the operational system volume control. In this recipe, we will create a sound volume control GUI that can be switched from a single volume bar to independent music and effects controls.

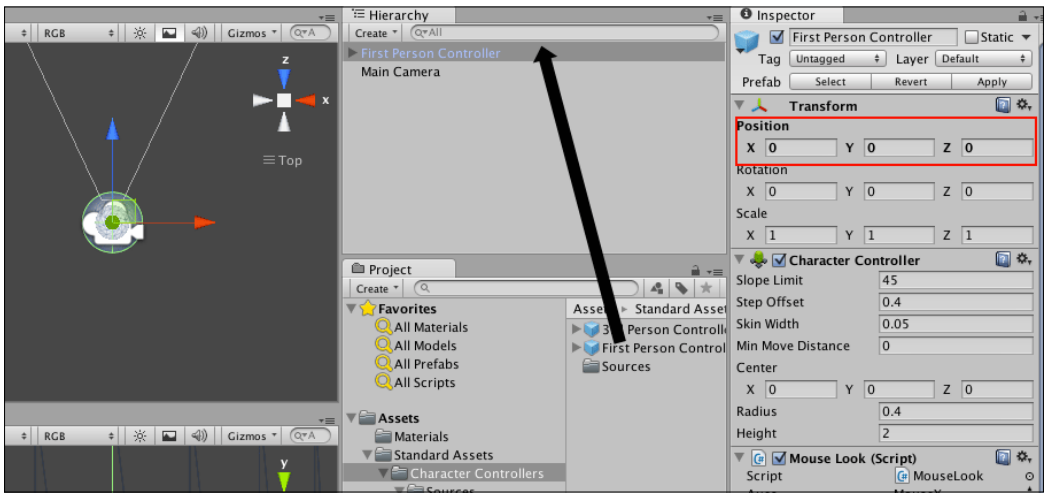
Getting ready

For this recipe, you'll need the `soundFX.wav` and `soundtrack.mp3` audio files, available in the `0423_06_02` folder.

How to do it...

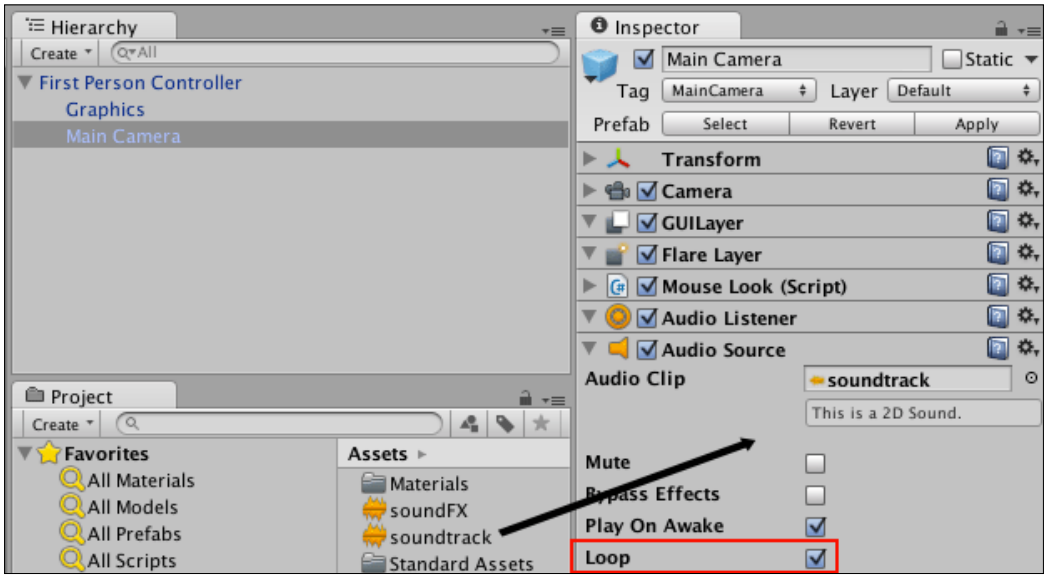
To add volume control sliders to your scene, follow these steps:

1. Import the required `soundtrack.mp3` audio file.
2. In the **Project** view, select the `soundtrack.mp3` file. Make sure the **3D Sound** option of the **Audio Importer** (in the **Inspector** view) is unselected. If not, unselect it.
3. Let's now import the `soundFX.wav` audio clip. This time, we will make sure to leave the **3D Sound** option checked.
4. Make sure the **First Person Controller** prefab is available in your project. You can do that by importing it from **Assets** | **Import Package...** | **Character Controller**.
5. Add the **First Person Controller** prefab to your scene by dragging it from the **Project** view to the **Hierarchy** view. Then, in the **Inspector** view, reset its position to **X: 0**, **Y:0**, **Z:0**, as shown in the following screenshot:

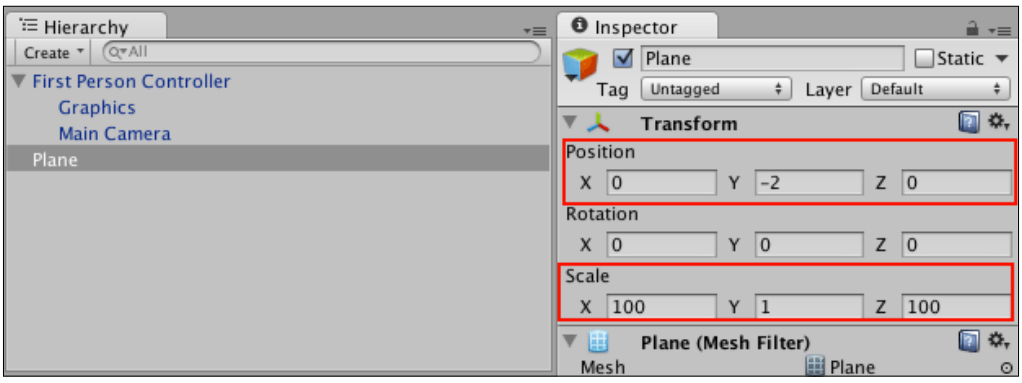


www.it-ebooks.info

6. Delete the original **Main Camera** from the scene.
7. Expand the **First Person Controller** object in the **Hierarchy** view and select the **Main Camera** child.
8. Drag the **soundtrack** audio clip file from the **Project** view to the bottom of the **Main Camera Inspector** view, adding it as an **Audio Source** component. Check the **Loop** option:

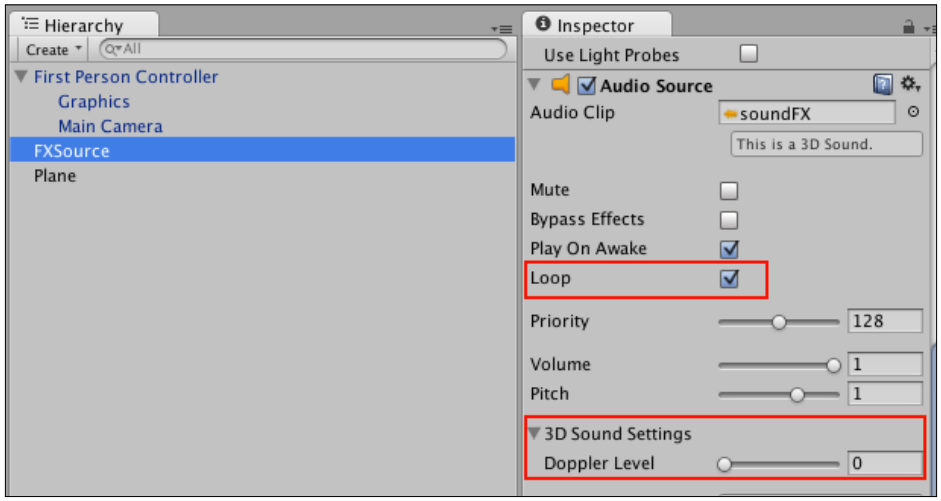


9. Using the **Create** drop-down of the **Hierarchy** view, add a **Plane** to your scene.
10. In the **Transform** component of the **Inspector** view, change the **Plane's Scale X** and **Z** values to **100**. Also, change its position to **X:0**, **Y: -2**, **Z:0**. The **Plane** object will be the ground for our scene:



www.it-ebooks.info

11. Using the **Create** drop-down of the **Hierarchy** view, add a Cube to your scene and rename it to `FXSource`. Set its position to **X:0**, **Y:0**, **Z:6**.
12. Drag the **soundFX** clip from the **Project** view to bottom of the **FXSource Inspector** view, adding it as an **Audio Source** component.
13. In the **Audio Source** component of the **FXSource** cube, check the **Loop** option and set **Doppler Level** to **0**:



14. Add a **Directional Light** to the scene.
15. In the **Project** view, create a new C# script and rename it `VolumeControl`.
16. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
[RequireComponent(typeof(AudioSource))]
public class VolumeControl : MonoBehaviour{
    bool separateSoundtrack = true;
    float minVolume = 0.0f;
    float maxVolume = 1.0f;
    float initialVolume = 1.0f;
    float soundtrackVolume = 1.0f;
    bool displaySliders = false;

    void Start(){
        if (separateSoundtrack){
            audio.ignoreListenerVolume = true;
        }
    }
    void Update(){
```


www.it-ebooks.info

```
        AudioListener.volume = initialVolume;
        if (separateSoundtrack){
            audio.volume = soundtrackVolume;
        }else{
            audio.volume = initialVolume;
        }
    }

    void OnGUI(){
        Event e = Event.current;
        if (e.type == EventType.KeyUp && e.keyCode == KeyCode.
Escape){
            displaySliders = !displaySliders;
        }
        if (displaySliders){
            if (!separateSoundtrack){
                GUI.Label(new Rect(10, 0, 100, 30), "Volume");
                initialVolume = GUI.HorizontalSlider(new Rect(10,
20, 100, 30), initialVolume, minVolume, maxVolume);
            }else{
                GUI.Label(new Rect(10, 0, 100, 30), "Sound FX");
                initialVolume = GUI.HorizontalSlider(new Rect(10,
20, 100, 30), initialVolume, minVolume, maxVolume);
                GUI.Label(new Rect(10, 40, 100, 30), "Music");
                soundtrackVolume = GUI.HorizontalSlider(new
Rect(10, 60, 100, 30), soundtrackVolume, minVolume, maxVolume);
            }
        }
    }
}
```

17. Save your script and attach it to the **Main Camera** by dragging it from the **Project** view to camera game object in the **Hierarchy** view.
18. Play the scene and hit *Esc* in your keyboard. You'll see the volume slidebars on the top-left of the game's viewport.

How it works...

By default, every sound in the scene has its volume controlled by the **Volume** parameter of the camera's **Audio Listener** component. With our script, we assign the **Volume** slidebar value to that parameter. Also, we create a separate volume bar for the `soundtrack.mp3` file, making it independent from the camera's **Audio Listener** component.

www.it-ebooks.info

There's more...

Here is some information on how to fine-tune and customize this recipe.

Using a single volume bar

If you need to simplify the volume controls even more, you can use a single volume bar by leaving the **Separate Soundtrack** option unchecked.

Bypassing the Esc key

If you want your volume bars to be displayed automatically, make sure to tick the checkbox for the parameter named **Display Sliders**.

See also

- The *Making a dynamic soundtrack* recipe.
- The *Pausing the game* recipe.

Simulating a tunnel environment with Reverb Zones

Once you have created your level's geometry, and the scene is looking just the way you want it to, you might want your sound effects to correspond to that look. Sound behaves differently depending on the environment it is projected, so it can be a good idea to make it reverberate accordingly. In this recipe, we will address this acoustic effect by using **Reverb Zones**.

Getting ready

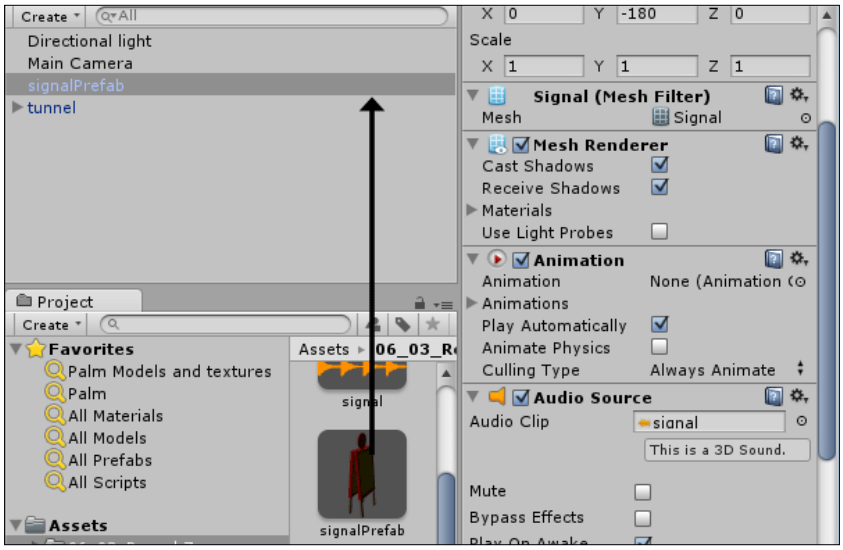
For this recipe, we have prepared a package containing a basic level named `reverbZoneLevel` and the signal prefab. The package is in the `0423_06_03` folder.

How to do it...

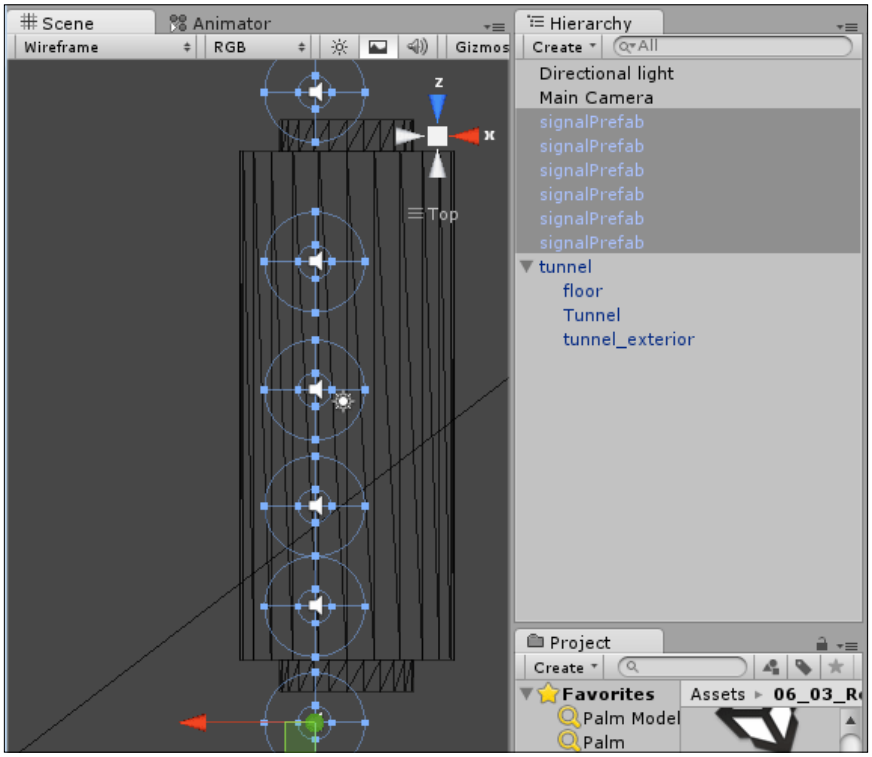
Follow these steps to simulate the sonic landscape of a tunnel:

1. Import the `reverbZones` package into your Unity project.
2. In the **Project** view, open the `reverbZoneLevel` level, inside the **06_03 ReverbZones** folder. This is a basic scene featuring a first-person camera and a tunnel.
3. Now drag **signalPrefab** from the **Project** view into the **Hierarchy** view. That should add a sound-emitting object to the scene. Place it in the center of the tunnel.

www.it-ebooks.info

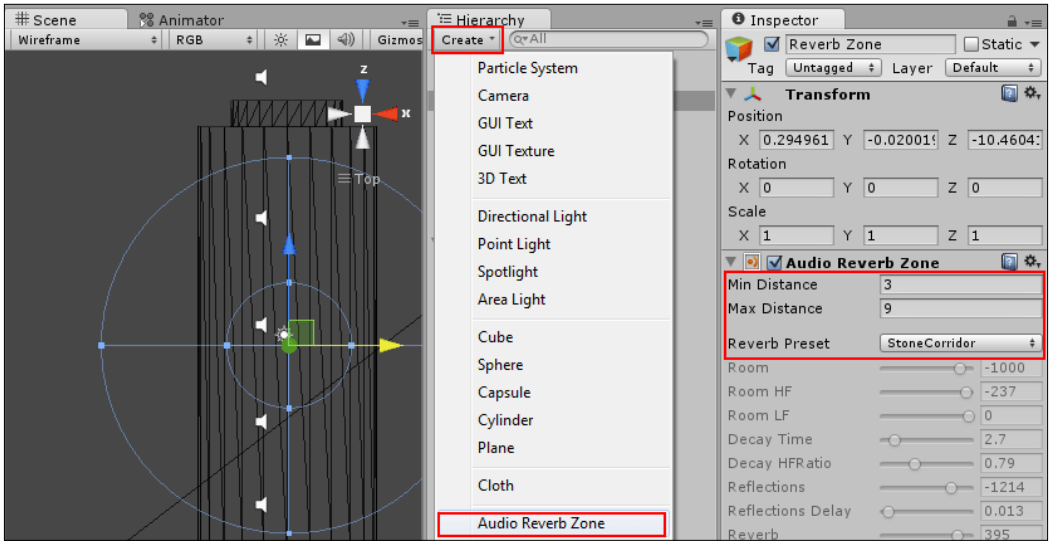


4. Make five copies of the **signalPrefab** game object and distribute them across the tunnel (leaving a copy just outside each entrance), as shown here:



www.it-ebooks.info

5. In the **Hierarchy** view, click **Create** to add an **Audio Reverb Zone** to the scene. Now place it in the center of the tunnel.
6. Select the **Reverb Zone** game object. In the **Inspector** view, change the **Reverb Zone** component parameters to these values: **Min Distance**: **3**; **Max Distance**: **9**; **Preset**: **StoneCorridor**:



7. Play the scene and walk through the tunnel. You should hear the audio reverberate when inside the **Reverb Zone** area.

How it works...

Once positioned, the **Audio Reverb Zone** applies an audio filter to all audio sources within its radius.

There's more...

Here are more options for you to try.

Attaching the Audio Reverb Zone component to audio sources

Instead of creating an **Audio Reverb Zone** game object, you could attach it to the sound emitting object (in our case, **signalPrefab**) as a component by navigating to **Component | Audio | Audio Reverb Zone**. In this case, the **Reverb Zone** would be individually set up around the object.

www.it-ebooks.info

Making your own Reverb settings

Unity comes with several Reverb presets. We have used **StoneCorridor**, but your scene could ask for something less intense (such as **Room**) or more radical (such as **Psychotic**). If those presets still won't be able to recreate the effect you have in mind, change it to **User** and edit the parameters as you wish.

Preventing the AudioClip from restarting if already playing

In a game there may be several different events that cause a sound to start playing. If the sound is already playing, then in almost all cases we don't wish to restart the sound. This recipe includes a test, so that an **AudioSource** component is only sent a `Play()` message if it is not currently playing.

Getting ready

Try this with any audio clip that is one second or longer in duration.

How to do it...

To prevent an audio clip from restarting, follow these steps:

1. Create an empty game object named `AudioObject`, and add an audio source component to this object.
2. Drag an audio clip file from the **Project** view to populate the **AudioClip** parameter of the **AudioSource** component of **AudioObject**.
3. Add the following script class to the **Main Camera**:

```
// file: AvoidSoundRestart.cs
using UnityEngine;

public class AvoidSoundRestart : MonoBehaviour{
  public AudioSource audioSource;

  private void OnGUI(){
    string statusMessage = "audio source - not playing";
    if(audioSource.isPlaying )
       statusMessage = "audio source - playing";

    GUILayout.Label( statusMessage );
```


www.it-ebooks.info

```
    bool buttonWasClicked = GUILayout.Button("send Play()
message");
    if( buttonWasClicked )
      PlaySoundIfNotPlaying();
  }

  private void PlaySoundIfNotPlaying(){
    if( !audioSource.isPlaying )
      audioSource.Play();
  }
}
```

4. With the **Main Camera** selected in the **Hierarchy** view, drag **AudioObject** into the **Inspector** for the public **AudioSource** variable.

How it works...

`AudioSource` components have a public readable property named `isPlaying`, which is a Boolean true/false flag indicating if the sound is currently playing. The `PlaySoundIfNotPlaying()` method includes an `if` statement ensuring that a `Play()` message is only sent to the `AudioSource` component if its `isPlaying` is false.

See also

- The *Waiting for audio to finish before auto-destructing an object* recipe.

Waiting for audio to finish before auto-destructing an object

An event may occur (such as an object pickup, or the killing of an enemy) that we wish to notify to the player by playing an audio clip and an associated visual object (such as an explosion particle system, or a temporary object in the location of the event). However, as soon as the clip has finished playing, we want the visual object to be removed from the scene. This recipe provides a simple way to link the ending of an audio clip that's playing with the automatic destruction of its parent `GameObject`.

Getting ready

Try this with any audio clip that is one second or longer in duration.

www.it-ebooks.info

How to do it...

To wait for the audio to finish before destroying an object, follow these steps:

1. Create an empty game object named `AudioObject`, and add an audio source component to this object.
2. Drag an audio clip file from the **Project** view to populate the **AudioClip** parameter of the **AudioSource** component of **AudioObject**, and deselect the component's **Play On Awake** checkbox.
3. Add the following script class to **AudioObject**:

```
// file: AudioDestructBehaviour.cs
using UnityEngine;
using System.Collections;

public class AudioDestructBehaviour : MonoBehaviour {
  private void Update()
  {
    if( !audio.isPlaying )
      Destroy(gameObject);
  }
}
```

4. Add the following script class to the **Main Camera**:

```
// file: PlayDestroyButtonGUI.cs
using UnityEngine;
using System.Collections;

public class PlayDestroyButtonGUI : MonoBehaviour{
 public AudioDestructBehaviour myAudioDestructObect;

 private void OnGUI(){
  bool playButtonWasClicked = GUILayout.Button("play");
  bool destroyButtonWasClicked = GUILayout.Button("play then
destroy");

  if( playButtonWasClicked ){
   myAudioDestructObect.audio.Play();
  }

  if( destroyButtonWasClicked ){
   myAudioDestructObect.audio.Play();
   myAudioDestructObect.enabled = true;
  }
 }
}
```


www.it-ebooks.info

5. With the **Main Camera** selected in the **Hierarchy** view, drag **AudioObject** into the **Inspector** view for the public **AudioSource** variable `myAudioDestructObect`.
6. With the **AudioObject** selected in the **Hierarchy** view, disable the scripted component `AutoDestructBehaviour` (uncheck the box by this component).

How it works...

The game object named **AudioObject** contains an **AudioSource** component, which stores and manages the playing of audio clips. **AudioObject** also contains a scripted component, which is an instance of the `AudioDestructBehaviour` class. When enabled, in every frame this object (via its `Update()` method) tests whether the audio source is not playing (`!audio.isPlaying`). As soon as the audio is found not to be playing the game object is destroyed. While the audio source is playing, then no action is taken.

The **Main Camera** scripted object `PlayDestroyButtonGUI` offers two buttons to the user, both buttons send a `Play()` message to the audio source component of the audio game object. However, when the **Play then destroy** button is clicked, the scripted component is also enabled. By enabling the scripted object, it means the logic in its `Update()` method will be tested each frame, and as soon as the audio clip has finished playing, then the parent game object will be destroyed.

See also

- The *Preventing the AudioClip from restarting if already playing* recipe.

Making a dynamic soundtrack

Dynamic soundtracks are the ones that change according to what is happening to the player in the game, musically reflecting that place or moment of the character's adventure. In this recipe, we will implement a soundtrack that changes when the player reaches specific targets. Also, we will have the option of fading the sound in and out.

Getting ready

For this recipe, we have prepared a basic level and some soundtrack audio files in `.ogg` format. They are contained inside the Unity package named `DynamicSoundtrack`, which can be found in the `0423_06_06` folder.

www.it-ebooks.info

How to do it...

To make a dynamic soundtrack, follow these steps:

1. Import the `DynamicSoundtrack` package into your Unity project. Also, import the `00_main`, `01_achievement`, and `02_danger` audio files.
2. Open the level named **SoundtrackScene**. It should include a basic terrain, a **3rd Person Controller** and three spheres named `Music Sphere`.
3. In the **Project** view, create a new C# script and name it `DynamicSoundtrack`.
4. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class DynamicSoundtrack : MonoBehaviour{
    public AudioClip[] clips;
    public int startingTrack = 0;
    private int currentTrack;
    private int nextTrack;
    private bool isFadingOut = false;
    private float fadeOutTime = 1.0f;
    private bool isFadingIn = false;
    private float fadeInTime = 1.0f;
    private bool waitSequence = true;
    private bool keepTime = false;
    private float targetVolume = 1.0f;
    private float oldVolume = 0.0f;
    private float fadeOutStart = 0.0f;
    private float fadeInStart = 0.0f;

    void Start(){
        audio.clip = clips[startingTrack];
        audio.Play();
        currentTrack = startingTrack;
    }

    void Update(){
        if (isFadingOut){
            if (audio.volume > 0){
                float elapsOut = Time.time - fadeOutStart;
                float indOut = elapsOut / fadeOutTime;
                audio.volume = oldVolume - (indOut * oldVolume);
            }else{
```


www.it-ebooks.info

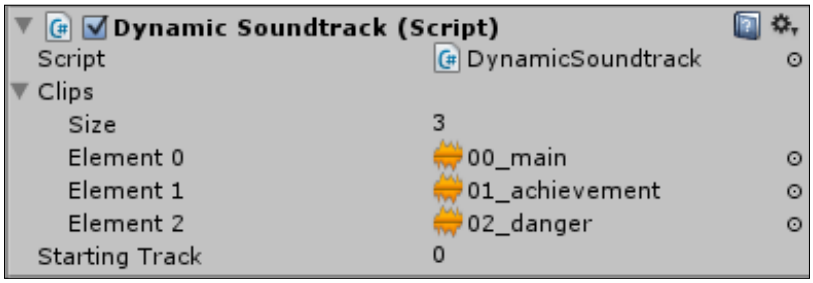
```
                isFadingOut = false;
                StartCoroutine(PlaySoundtrack());
            }
        }

        if (isFadingIn){
            if (audio.volume < targetVolume){
                float elapsIn = Time.time - fadeInStart;
                float indIn = elapsIn / fadeInTime;
                audio.volume = indIn;
            }else{
                audio.volume = targetVolume;
                isFadingIn = false;
            }
        }
    }

    public void ChangeSoundtrack(int newClip, bool
waitForSequence, bool keepPreviousTime, float trackVolume, float
fadeIn, float fadeOutPrevious){
        nextTrack = newClip;
        waitSequence = waitForSequence;
        keepTime = keepPreviousTime;
        targetVolume = trackVolume;
        fadeInTime = fadeIn;

        if (newClip != currentTrack){
            currentTrack = newClip;
            if (fadeOutPrevious != 0){
                oldVolume = audio.volume;
                fadeOutStart = Time.time;
                fadeOutTime = fadeOutPrevious;
                isFadingOut = true;
            }else{
                StartCoroutine(PlaySoundtrack());
            }
        }
    }
    IEnumerator PlaySoundtrack(){
        if (waitSequence)
            yield return new WaitForSeconds(audio.clip.length -
((float)audio.timeSamples / (float)audio.clip.frequency));

if(fadeInTime !=0){
```


www.it-ebooks.info

```
        audio.volume = 0;
        fadeInStart = Time.time;
        isFadingIn = true;
            }
            float StartingPoint = 0.0f;
            if (keepTime)
                StartingPoint = audio.timeSamples;

            audio.clip = clips[nextTrack];
            audio.timeSamples = Mathf.RoundToInt(StartingPoint);
            audio.Play();
        }
}
```

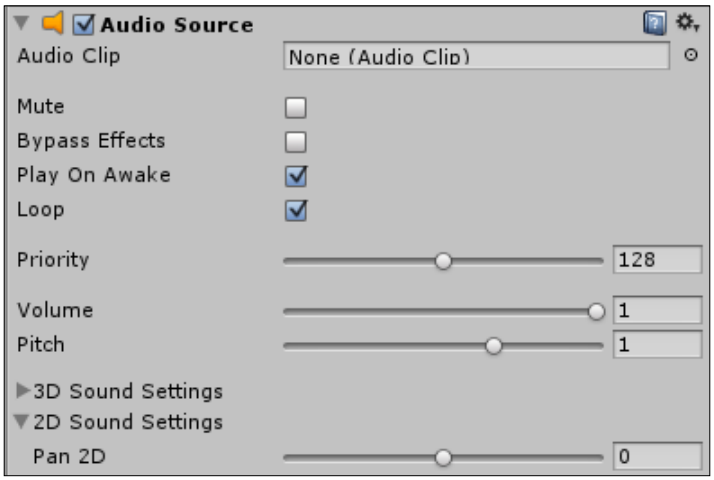
In case you are wondering why we are using `timeSamples` instead of `time`, it's because the former is more accurate when working with compressed audio files. To find out its actual time, we used the expression `audio.timeSamples / audio.clip.frequency`. As this is not currently documented in Unity's Scripting Reference, we thank audio engineer Aldo Naletto for this tip (and reviewer Peter Bruun for reminding us of using `float`, for more precision).

5. Save your script and attach it to the **Main Camera** by dragging it from the **Project** view to the **Main Camera** game object in the **Hierarchy** view.
6. Select the **Main Camera** and, in the **Inspector** view, access the **Dynamic Soundtrack** component and change the **Size** parameter of the **Clips** variable to **3**. Then, drag the `0_Main`, `1_Achievement`, and `2_Danger` sound files from the **Project** view into the appropriate slots. Also, type in `0` into the slot named **Starting Track**:



www.it-ebooks.info

7. Now, access the **Audio Source** component and make sure the **Loop** option is checked:

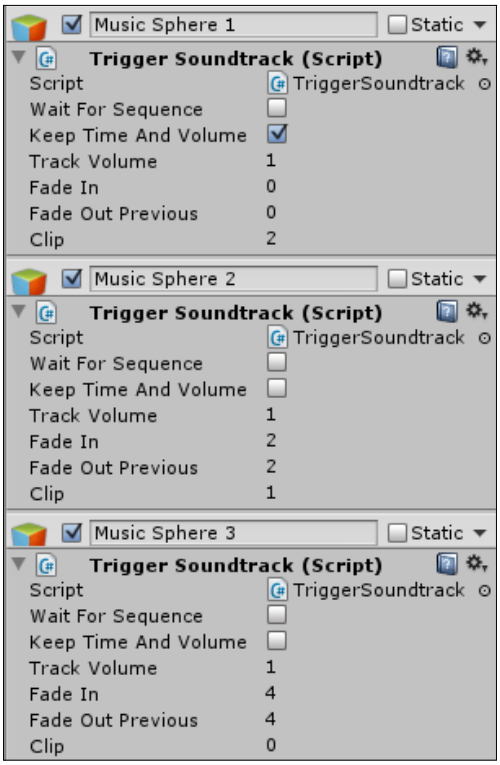


8. In the **Project** view, create a new C# script and name it `TriggerSoundtrack`.
9. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class TriggerSoundtrack : MonoBehaviour{
    public bool waitForSequence = true;
    public bool keepTimeAndVolume = false;
    public float trackVolume = 1.0f;
    public float fadeIn = 0.0f;
    public float fadeOutPrevious = 0.0f;
    public int clip;
    private DynamicSoundtrack soundtrack;

    void Awake(){
        soundtrack = Camera.main.GetComponent<DynamicSoundtra
ck>();
    }
    void OnTriggerEnter(Collider other){
        if (other.gameObject.CompareTag("Player"))
            soundtrack.ChangeSoundtrack(clip, waitForSequence,
keepTimeAndVolume, trackVolume, fadeIn, fadeOutPrevious);
    }
}
```


www.it-ebooks.info

10. Save your script and attach it to the each one of the music spheres by dragging it from the **Project** view to the **Main Camera** game object in the **Hierarchy** view.
11. Select each **Music Sphere** object and, in the **Inspector** view, change the **Trigger Soundtrack** parameters, as shown in the following screenshot:



12. Play your scene and direct the character towards each **Music Sphere** object. The background music will change accordingly.

How it works...

We have created two different scripts. The one attached to the **Main Camera**, **DynamicSoundtrack**, is responsible for keeping a list of the audio files that make up the entire soundtrack for the level. It also contains all of the functions that control the audio playback, volume transition, and so on. The second one, **TriggerSoundtrack**, is attached to the **Music Sphere** objects and triggers soundtrack changes based on the preferences expressed in that component's parameters. They are:

- **Wait For Sequence**: Leave this option checked if you want to wait until the end of the previous audio clip before playing the new part.

www.it-ebooks.info

- **Keep Time And Volume**: Leave it checked to start a new audio clip from the same point where the previous clip was at. This also keeps the volume level from the previous clip.
- **Track Volume**: The volume for the new audio clip (from 0.0 to 1.0).
- **Fade In**: The amount of time in seconds it will take for the new audio clip's volume to fade in.
- **Fade Out Previous**: The amount of time in seconds it will take for the previous audio clip's volume to fade out.

There's more...

Here is some information on how to fine-tune and customize this recipe.

Hiding the triggers

If having milestone objects as triggers feels too obvious for you and your players, you can always make it invisible by disabling the **Mesh Renderer** component.

Dealing with audio file formats and compression rates

To avoid loss of audio quality, you should import your sound clips using the appropriate file format, depending on your target platform. If you are not sure which format to use, please check out Unity's documentation on this subject at `http://docs.unity3d.com/Documentation/Manual/AudioFiles.html`.

Using 2D sound

To make the sound volume and balance independent of the audio source position, make sure your audio clip is not set up as **3D Sound** (you can check it out at **Import Settings** in the **Inspector** view, by selecting the file in the **Project** view).

www.it-ebooks.info

7

Working with External Resource Files and Devices

In this chapter, we will cover:

- Loading external resource files – by Unity Default Resources
- Loading external resource files – by manually storing files in Unity's Resources folder
- Loading external resource files – by downloading files from the Internet
- Saving and loading player data – using static properties
- Saving and loading player data – using PlayerPrefs
- Saving screenshots from the game
- Control characters in Unity with the Microsoft Kinect using the Zigfu samples
- Animating your own characters with the Microsoft Kinect controller
- Homemade mocap by storing movements from the Microsoft Kinect controller
- Setting up a leaderboard using PHP/MySQL

Introduction

External data can make your game better in many ways: it might add renewable content, help file organization, and allow user preferences to be set. Also, it can make Unity talk with other peripherals. In every way, it can deliver a richer experience to players and developers alike. There are several different places from which external files can be stored and read from.

In this chapter we present a wide range of uses for external data.

www.it-ebooks.info

Loading external resource files – by Unity Default Resources

In this recipe, we will load an external image file and display it on the screen using the Unity **Default Resources** file (a library created at the time the game is compiled).

This method is perhaps the simplest way to store and read external resource files. However, it is only appropriate when the contents of the resource files will not change after compilation since the contents of these text files are combined and compiled into the `resources.assets` file.

The `resources.assets` file can be found in the `Data` folder for a compiled game.

Getting ready

In the `0423_07_01` folder we have provided the following for this recipe: an image file, a text file, and an audio file in Ogg format:

- `externalTexture.jpg`
- `cities.txt`
- `soundtrack.ogg`

How to do it...

1. In the **Project** window, create a new folder and rename it `Resources`.
2. Import the `externalTexture.jpg` file and place it into the `Resources` folder.
3. Add the following C# script to the **Main Camera**:

```
// file: ReadDefaultResources.cs
using UnityEngine;
using System.Collections;

public class ReadDefaultResources : MonoBehaviour {
  public string fileName = "externalTexture";
  private Texture2D externalImage;

  private void Start () {
    externalImage = (Texture2D)Resources.Load(fileName);
  }

  private void OnGUI() {
```


www.it-ebooks.info

```
        GUILayout.Label(externalImage);
      }
    }
```

4. Play the scene. The texture will be loaded and displayed on the screen.
5. If you have another image file, put a copy into in the `Resources` folder, then in the **Inspector** view, change the public filename to the name of your image file and play the scene again. The new image should now be displayed.

How it works...

The `Resources.Load(fileName)` statement makes Unity look inside its compiled `resources.assets` project data file for the contents of file named `externalTexture`. The contents are returned as a texture image, which is stored in the `externalImage` variable. Our `OnGU()` method displays `externalImage` as `Label()`.

The filename string passed to `Resources.Load()` does *not* include the file extension (such as `.jpg` or `.txt`).

There's more...

Here are some details you don't want to miss.

Loading text files with this method

You can load external text files using the same approach. The private variable needs to be a String (to store the text file contents). The `Start()` method uses a temporary `TextAsset` object to receive the text file contents, and the `text` property of this object contains the String contents to be stored in the private variable:

```
public class ReadDefaultResources : MonoBehaviour {
  public string fileName = "textFileName";
  private string textFileContents;

  private void Start () {
    TextAsset textAsset = (TextAsset)Resources.Load(fileName);
    textFileContents = textAsset.text;
  }

  private void OnGUI() {
    GUILayout.Label(textFileContents);
  }
}
```


www.it-ebooks.info

Loading and playing audio files with this method

You can load external audio files using the same approach. The private variable needs to be an `AudioClip` object:

```
// file: ReadDefaultResources.cs
using UnityEngine;
using System.Collections;

[RequireComponent (typeof (AudioSource))]

public class ReadDefaultResources : MonoBehaviour {
  public string fileName = "soundtrack";
  private AudioClip audioFile;

  void  Start (){
    audio.clip = (AudioClip)Resources.Load(fileName);
    if(!audio.isPlaying && audio.clip.isReadyToPlay)
      audio.Play();
  }

  private void OnGUI () {
    if(GUILayout.Button("Play")){ audio.Play(); }
    if(GUILayout.Button("Pause")){ audio.Pause(); }
    if(GUILayout.Button("Stop")){ audio.Stop(); }  }
}
```

See also

- The *Loading external resource files – by manually storing files in Unity's Resources folder* recipe.
- The *Loading external resource files – by downloading files from the Internet* recipe.

Loading external resource files – by manually storing files in Unity's Resources folder

At times the contents of external resource files may need to be changed after the game is compiled. Hosting resource files on the Web may not be an option. There is a method of manually storing and reading files from the `Resources` folder of the compiled game. This allows for those files to be changed after the game's compilation.

www.it-ebooks.info

This technique only works when you compile to a Windows or Mac *standalone executable*. It will not work for Web Player builds, for example.

Getting ready

The `0423_07_01` folder provides the `externalTexture.jpg` texture image that you can use for this recipe.

How to do it...

1. Add the following C# script to the **Main Camera**:

```
// file: ReadManualResourceImageFile.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class ReadManualResourceImageFile : MonoBehaviour {
  private string fileName = "externalTexture.jpg";
  private string url;
  private Texture2D externalImage;

  private void Start () {
     url = "file:" + Application.dataPath;
     url = Path.Combine(url, "Resources");
     url = Path.Combine(url, fileName);

    StartCoroutine( LoadWWW() );
  }

  private void OnGUI() {
    GUILayout.Label ( "url = " + url );
    GUILayout.Label ( externalImage );
  }

  private IEnumerator LoadWWW(){
    yield return 0;
    WWW www = new WWW (url);
    yield return www;
      externalImage = www.texture;
  }
}
```


www.it-ebooks.info

2. Build your (Windows or Mac) standalone executable.
3. Copy the `externalTexture.jpg` image into your standalone's `Resources` folder.

You will need to place the files in the `Resources` folder manually after every compilation.

WINDOWS and LINUX: When you create a Windows or Linux standalone executable, there is also a `_Data` folder created with the executable application file. The `Resources` folder can be found inside this data folder.

MAC: A Mac standalone application executable looks like a single file, but it is actually a Mac OS `package` folder. Right-click the executable file and select **Show Package Contents**. You will then find the standalone's `Resources` folder inside the `Contents` folder.

4. Run your standalone game application, and the image should be displayed.

How it works...

Note the need to use the `System.IO` package for this recipe.

When the executable runs, the `WWW` object spots that the URL starts with the word "file:" so Unity attempts to find the external resource file in its `Resources` folder and then load its contents.

There's more...

Here are some details you don't want to miss.

Avoiding cross-platform problems with Path.Combine() rather than "/" or "\"

The filepath folder separator character is different for Windows and Mac filesystems ("\" backslash for Windows, "/" forward slash for the Mac). However, Unity knows which kind of standalone you are compiling your project into, therefore the `Path.Combine()` method will insert the appropriate separator slash character to form the file URL that is required.

Loading a text file using this approach

To use this technique to load an external text file, you'll need a private string variable:

```
private string textFileContents = "(still loading file ...)";
```


www.it-ebooks.info

Replace your `LoadWWW()` method with one that extracts the text of the loaded resource and stores it in your private string variable:

```
private IEnumerator LoadWWW(){
  yield return 0;
  WWW www = new WWW (url);
  yield return textFile;
    textFileContents = www.text;
}
```

Then change `OnGUI()` to display the string as a label:

```
private void OnGUI() {
  GUILayout.Label ( "url = " + url );
  GUILayout.Label ( textFileContents );
}
```

WWW and resource contents

The `WWW` class defines several different properties and methods to allow downloaded media resource file data to be extracted into appropriate variables for use in the game. The most useful of these include:

- `.text`: A read-only property returning web data as a string
- `.texture`: A read-only property returning web data as a `Texture2D` image
- `.GetAudioClip()`: A method that returns web data as an `AudioClip` object

For more information about the Unity `WWW` class, see `http://docs.unity3d.com/Documentation/ScriptReference/WWW.html`.

See also

- The *Loading external resource files – by Unity Default Resources* recipe.
- The *Loading external resource files – by downloading files from the Internet* recipe.

Loading external resource files – by downloading files from the Internet

One way to store and read text file data is to store the text files on the Web. In this recipe, the contents of a text file for a given URL is downloaded, read, and then displayed.

www.it-ebooks.info

Getting ready

For this recipe you will need to have access to files on a web server. If you run a local web server such as Apache, or have your own web hosting, then you could use the files in the `0423_07_01` folder and the corresponding URL. Otherwise, you may find the following URLs useful since they are the web locations of an image file (a Packt logo) and a text file (an "ASCII-art" badger picture):

- `www.packtpub.com/sites/default/files/packt_logo.png`
- `www.ascii-art.de/ascii/ab/badger.txt`

How to do it...

1. Add the following C# script to the **Main Camera**:

```
// file: ReadWebImageTexture.cs
using UnityEngine;
using System.Collections;

public class ReadWebImageTexture : MonoBehaviour {
  public string url = "http://www.packtpub.com/sites/default/
files/packt_logo.png";
  private Texture2D externalImage;

  private void Start () {
    StartCoroutine( LoadWWW() );
  }

  private void OnGUI() {
    GUILayout.Label ( "url = " + url );
    GUILayout.Label ( externalImage );
  }

  private IEnumerator LoadWWW(){
    yield return 0;
    WWW imageFile = new WWW (url);
    yield return imageFile;
      externalImage = imageFile.texture;
  }
```

2. Play the scene. Once downloaded, the contents of the image file will be displayed.

www.it-ebooks.info

How it works...

When the game starts, our `Start()` method starts the `LoadWWW()` co-routine method. A co-routine is a method that can keep on running in the background without halting or slowing down the other parts of the game and the frame rate. The `yield` statement indicates that once a value can be returned for `imageFile`, the remainder of the method can be executed, that is, until the file has finished downloading, no attempt should be made to extract the `texture` property of the `WWW` object variable.

For each frame our `OnGUI()` method displays the contents of `url` string variable and attempts to display the `externalImage` texture image.

See also

- The *Loading external resource files – by Unity Default Resources* recipe.
- The *Loading external resource files – by manually storing files in Unity's Resources folder* recipe.

Saving and loading player data – using static properties

Keeping track of the player's progress and user settings during a game is vital to give your game a greater feel of depth and content. In this recipe, we will learn how to make our game "remember" the player's name and score between different levels (scenes). Also, we will give the player the option of erasing his data.

Getting ready

We have included a complete project named `BlueSpheres` in the `0423_07_04` folder. In order to follow this recipe, make a copy of that folder as your starting point.

How to do it...

To save and load player data, follow these steps:

1. Open the `BlueSpheres` project and make yourself familiar with the game by playing it a few times and examining the contents of the scenes. The game starts on the `menu_start` scene, inside the `Scenes` folder.

www.it-ebooks.info

2. Create a new `Player` C# script with the following code:

```
// file: Player.cs
using UnityEngine;
using System.Collections;

public class Player : MonoBehaviour {
  public static string username = null;
  public static int score = -1;

  public static void DeleteAll(){
    username = null;
    score = -1;
  }
}
```

3. Replace the contents of the `MenuStart` C# script with the following code:

```
// file: MenuStart.cs
using UnityEngine;
using System.Collections;

public class MenuStart : MonoBehaviour
{
  const string DEFAULT_PLAYER_NAME = "PLAYER_NAME";
  private string playerNameField = DEFAULT_PLAYER_NAME;

  private void OnGUI() {
    string rules = "Easiest Game Ever -- Click the blue spheres to
advance.";
    GUILayout.Label(rules);

    if(Player.username != null)
      WelcomeGUI();
    else
      CreatePlayerGUI();
  }

  private void WelcomeGUI() {
    string welcomeMessage = "Welcome, " + Player.username + ". You
currently have " + Player.score + " points.";
        GUILayout.Label(welcomeMessage);

    bool playButtonClicked = GUILayout.Button("Play");
    bool eraseButtonClicked = GUILayout.Button("Erase Data");
```


www.it-ebooks.info

```
    if( playButtonClicked )
      Application.LoadLevel(1);

    if( eraseButtonClicked )
      ResetGameData();
  }

  private void ResetGameData() {
    Player.DeleteAll();
    playerNameField = DEFAULT_PLAYER_NAME;
  }

  private void CreatePlayerGUI() {
      string createMessage = "Please, insert your username below
and click on Create User";
    GUILayout.Label(createMessage);

      playerNameField = GUILayout.TextField(playerNameField, 25);

      bool createUserButtonClicked= GUILayout.Button("Create
User");

    if( createUserButtonClicked ){
      Player.username = playerNameField;
      Player.score = 0;
    }
  }
}
```

4. Replace the contents of the `SphereClick` C# script with the following code:

```
// file: SphereClick.cs
using UnityEngine;
using System.Collections;

public class SphereClick : MonoBehaviour
{
  private void OnGUI(){
    GUILayout.Label( "Score: " + Player.score );
  }

  private void OnMouseDown() {
    if( gameObject.CompareTag("blue") )
      Player.score += 50;
```


www.it-ebooks.info

```
    Destroy(gameObject);
    GotoNextLevel();
  }

  private void GotoNextLevel() {
    int level = Application.loadedLevel + 1;
    Application.LoadLevel(level);
  }
}
```

5. Replace the `OnGUI()` method of the `MenuEnd` C# script with the following code:

```
private void OnGUI() {
  GUILayout.Label("Congratulations " + Player.username);
  GUILayout.Label("You have finished the game, score = " + Player.
score);

  bool mainMenuButtonClicked = GUILayout.Button("Main Menu");
  if( mainMenuButtonClicked )
          Application.LoadLevel(0);
  }
```

6. Save your scripts and play the game. As you progress from level (scene) to level, you should find that the score and player's name are "remembered" until you quit the application.

How it works...

The `Player` class uses static (class) properties `username` and `score` to store the current player's name and score. Since these are public static properties, any object from any scene can access these values (static properties are "remembered" from scene to scene). This class also provides the `DeleteAll()` public static method that resets `username` to `null` and `score` to `-1`.

The `MenuStart` class tests the value of `Player.username`. If it is `null`, then a textbox and button are provided to allow the user to enter a new username; which is then stored in `Player.username`. Once the value in `Player.username` is no longer `null`, the user is offered a button to start playing the game, or to erase the username and any score currently stored in `Player` class.

The three game-playing scenes now include a `GUI` method to display the current value in the `Player.score` property, which has `50` added to it each time a sphere tagged `blue` is clicked.

The `MenuEnd` class congratulates the player by name (retrieving the name from `Player.username`), and tells them their score (from `Player.score`).

www.it-ebooks.info

See also

- The *Saving and loading player data – using PlayerPrefs* recipe.

Saving and loading player data – using PlayerPrefs

While the previous recipe illustrates how static properties allow a game to "remember" values between different scenes, those values are "forgotten" once the game application has quit. Unity provides the `PlayerPrefs` feature to allow a game to store and retrieve data between different game-playing sessions.

Getting ready

This recipe builds upon the previous recipe. In case you haven't completed the previous recipe, we have included a complete project named `BlueSpheres-static` in the `0423_07_05` folder. In order to follow this recipe, make a copy of that folder as your starting point.

How to do it...

To save and load player data, follow these steps:

1. Open the `BlueSpheres-static` project and delete the `Player` C# script.
2. Edit the `MenuStart` C# script by replacing the `OnGUI()` method with the following code (the `if` statement is to be changed):

```
private void OnGUI() {
string rules = "Easiest Game Ever -- Click the blue spheres to add
points and advance levels.";
GUILayout.Label(rules);

  if(PlayerPrefs.HasKey("username"))
    WelcomeGUI();
  else
    CreatePlayerGUI();
}
```

3. Edit the `MenuStart` C# script by replacing the `ResetGameData()` method with the following code:

```
private void ResetGameData() {
  PlayerPrefs.DeleteAll();
  playerNameField = DEFAULT_PLAYER_NAME;
}
```


www.it-ebooks.info

4. Edit the `MenuStart` C# script by replacing the first line of the `WelcomeGUI()` method with the following:

```
string welcomeMessage = "Welcome, " + PlayerPrefs.
GetString("username") + ". You currently have " + PlayerPrefs.
GetInt("score") + " points.";
```

5. Also, locate the lines:

```
Player.username = playerNameField;
Player.score = 0;
```

   And replace them with:

```
PlayerPrefs.SetString("username", playerNameField);
PlayerPrefs.SetInt("score", 0);
```

6. Now edit the `SphereClick` C# script by replacing the `ONGUI()` and `OnMouseDown()` methods with the following code:

```
private void OnGUI(){
  GUILayout.Label( "Score: " + PlayerPrefs.GetInt("score") );
}

private void OnMouseDown() {
  if( gameObject.CompareTag("blue") ){
    int newScore = 50 + PlayerPrefs.GetInt("score");
    PlayerPrefs.SetInt("score", newScore);
  }

  Destroy(gameObject);
  GotoNextLevel();
}
```

7. Edit the `MenuEnd` C# script by replacing the first two lines of the `OnGUI()` method with the following:

```
GUILayout.Label("Congratulations " + PlayerPrefs.
GetString("username"));

GUILayout.Label("You have finished the game, score = " +
PlayerPrefs.GetInt("score"));
```

8. Save your scripts and play the game. Close Unity and then restart the application. You should find that the player's name, level, and score are now kept between game sessions.

www.it-ebooks.info

How it works...

All we needed to do was to change our code to take advantage of Unity's `PlayerPrefs Runtime` class. This class is capable of storing and accessing information (`String`, `Int`, and `Float` variables) in the user's machine. Values are stored in a "plist" file (Mac) or the registry (Windows), in a similar way to web browser "cookies", and therefore "remembered" between game application sessions.

For more information on `PlayerPrefs`, see Unity's online documentation at `http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html`.

See also

- The *Saving and loading player data – using static properties* recipe.

Saving screenshots from the game

In this recipe, we will learn how to take in-game snapshots and save them to an external file. Better yet, we will make it possible to choose between two different methods.

This technique only works when you compile to a Windows or Mac *standalone executable*. It will not work for Web Player builds, for example.

Getting ready

In order to follow this recipe, please import the `screenshots` package, available in the `0423_07_06` folder, into your project. This package includes a basic terrain and a camera that can be rotated via a mouse.

How to do it...

To save screenshots from your game, follow these steps:

1. Import the `screenshots` package and open the `screenshotLevel` scene.
2. Add the following C# script to the **Main Camera**:

```
// file: TakeScreenshot.cs
using UnityEngine;
using System.Collections;
using System;
```

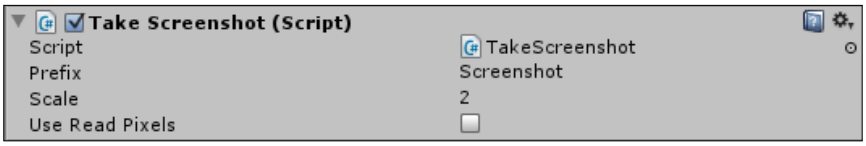

www.it-ebooks.info

```
using System.IO;
public class TakeScreenshot : MonoBehaviour {
  public string prefix = "Screenshot";
  public int scale = 1;
  public bool useReadPixels = false;
  private Texture2D texture;

  void  Update (){
   if (Input.GetKeyDown (KeyCode.P))
          StartCoroutine(TakeShot());
  }

  IEnumerator  TakeShot (){
    string date = System.DateTime.Now.ToString("_d-MMM-yyyy-HH-mm-
  ss-f");
    int sw = Screen.width;
    int sh = Screen.height;
    Rect sRect = new Rect(0,0,sw,sh);

    if (useReadPixels){
      yield return new WaitForEndOfFrame();
      texture = new Texture2D (sw,sh,TextureFormat.RGB24,false);
      texture.ReadPixels(sRect,0,0);
      texture.Apply();
      byte[] bytes= texture.EncodeToPNG();
      Destroy (texture);
      File.WriteAllBytes(Path.GetDirectoryName(Application.
  dataPath) + Path.DirectorySeparatorChar + prefix + date +
  ".png", bytes);

    } else {
       Application.CaptureScreenshot(prefix + date + ".png",
  scale);
     }
  }
}
```

3. Save your script and attach it to the **Main Camera** game object by dragging it from the **Project** view to the **Main Camera** game object in the **Hierarchy** view.
4. Access the **Take Screenshot** component. Set **Scale** to **2** and leave **Use Read Pixels** unchecked.

If you want your image file's name to start with something other than **Screenshot**, change it in the **Prefix** field.

www.it-ebooks.info



5. Play the scene. A new screenshot, with twice the original size, will be saved in your project folder every time you press *P*.

Please note that depending on the method used the file will be saved on a specific location within your game's folder. Mac users will have to right-click the executable file and select **Show Package Contents** in order to locate the file among the game data.

How it works...

Once the script has detected that the *P* key has been pressed, the screen is captured and stored as an image file in the same folder where the executable is. Unless **Use Read Pixels** is selected, the script will call a built-in Unity function called `CaptureScreenshot()`, which is capable of scaling up the original screen size (in our case, based on the `Scale` variable of our script).

There's more...

We have included the **Use Read Pixel** option as a demonstration of how to save your images to disk without using Unity's `CaptureScreenshot()` function. One advantage of this method is that it can be adapted to capture and save only a portion of the screen. The `Scale` variable from our script will not affect it needs, though.

Control characters in Unity with the Microsoft Kinect using the Zigfu samples

The Microsoft Kinect human motion controller offers an exciting way for players to interact with games and control their game characters. Some of the original developers of the software used in the Kinect started a company called **Zigfu**, and they now offer a Unity-Kinect development kit. This is a straightforward way to make Unity games that can use the Microsoft Kinect motion controller.

At the time of writing, an unlimited time free trial is available to download. The Zigfu free trial is like Unity free since it can be used for non-commercial projects and includes a watermark in projects.

www.it-ebooks.info

Be patient!

When you run a Kinect-device application, you will have to wait for 10-20 seconds for your computer to establish a link with the Kinect, and for the Kinect to detect a person standing in front of the camera. Learn more about the Microsoft Kinect at `www.xbox.com/KINECT`.

Getting ready

Go to the Zigfu website's downloads sections, and locate the Unity ZDK package (**Zigfu Development Kit**). At the time of writing, the ZDK Unity package is named `ZDK_Unity35_.99f_trial.unitypackage`, and can be found at `http://zigfu.com/en/zdk/unity3d/`.

How to do it...

1. Create a new Unity project and import the ZDK package.
2. You should now see two folders appear in your **Project** panel: **Standard Assets** and **Zigfu**.

You can ignore any warning messages about inconsistent line endings.

3. Open the **AvatarFrontfacing** scene, you'll find it in the **SampleScenes** folder inside the **Zigfu** folder. You should now see an avatar in the **Scene** and **Game** panels:

www.it-ebooks.info

4. Plug in your Kinect camera, and ensure the Kinect power supply is connected and switched on.
5. Run the scene.
6. After taking a few seconds to open communication between the sensor and Unity, you should then be able to control the avatar through your own body movements.

7. You'll also see the "radar" display (top-right, white square), indicating the position of each user relative to the camera, and the raw point cloud display (bottom-right, yellow-black square).

How it works...

The sample scene contains a character, and also all the scripted Zigfu objects connected to this character, so that when the Zigfu objects receive movement event messages from the Kinect camera, these movements are passed on to move and rotate the appropriate parts of the visible character.

See also

- The *Animating your own characters with the Microsoft Kinect controller* recipe.
- The *Homemade mocap by storing movements from the Microsoft Kinect controller* recipe.

www.it-ebooks.info

Animating your own characters with the Microsoft Kinect controller

Perhaps the best way to understand how the Zigfu objects and event messages can be used in your own Unity projects is to build a basic figure out of cubes, and associate particular cubes with particular skeleton objects. In this recipe, you'll build part of a block character from cubes (a head, left shoulder, elbow, and hand), and use Zigfu scripts to be able to control this arm when the game runs with the Kinect controller attached.

Getting ready

Please note that the scripts you'll be using in this recipe can be found in the following folders of the **Project** panel once you have imported the ZDK Unity package:

- `Zigfu/Scripts/UserEngagers/`
- `Zigfu/Scripts/UserControls/`
- `Zigfu/SampleScenes/Scripts/`
- `Standard Assets/Scripts/CameraScripts/`

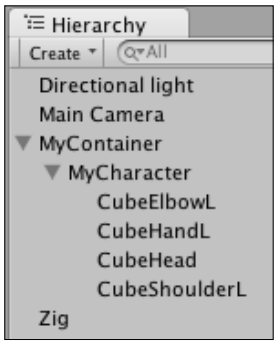
How to do it...

1. Create a new Unity project and import the ZDK package.
2. Add **Directional Light**.
3. Create an empty game object and name it `MyContainer`.
4. Create an empty game object named `MyCharacter`, and in the **Hierarchy** view, drag this object to become a child of **MyContainer**.
5. Create the following cubes, and in the **Hierarchy** view, drag these objects to become a children of **MyCharacter**:
    - A cube named `CubeHead` at (0,0,0) sized (0.2, 0.2, 0.2)
    - A cube named `CubeShoulderL` at (0.2, -0.2, 0) sized (0.2, 0.2, 0.2)
    - A cube named `CubeElbowL` at (0.5, -0.2, 0) sized (0.2, 0.2, 0.2)
    - A cube named `CubeHandL` at (0.8, -0.2, 0) sized (0.2, 0.2, 0.2)

The positions of these cubes don't matter since their positions in the game will be controlled by the head/arm positions detected by the Kinect sensor. However, it can be handy to lay them out in an approximation of a block-person to avoid getting confused when adding new cubes.

www.it-ebooks.info

6. Create an empty game object and name it **Zig**:



7. Add to **Main Camera** the **SmoothFollow** scripted component, set its **Distance** variable to **4**, and with **Main Camera** selected, in the **Inspector** view drag **CubeHead** into the public variable slot for **Target** for the **SmoothFollow** scripted component.
8. Add the **Zig** scripted component to **Zig**.
9. Add **ZigEngageSingleUser** scripted component to **Zig**.
10. While ensuring **Zig** is selected, in the **Inspector** view for the **ZigEngageSingleUser** component drag **MyCubeCharacter** over the **Engaged Users** public variable (its size should increase from 0 to 1, and **MyCharacter** should be listed as **Element 0**).
11. Add the **ZigSkeleton** scripted component to **MyCharacter**. In the **Inspector** view, ensure the following options are ticked (checked):
    - **Update Joint Positions**
    - **Update Root Positions**
    - **Update Orientation**
12. With **MyCharacter** selected, drag the following cubes in the **Inspector** view:
    - **CubeHead** into the public variable slot for **Head**
    - **CubeShoulderL** into the public variable slot for **Left Shoulder**
    - **CubeElbowL** into the public variable slot for **Left Elbow**
    - **CubeHandL** into the public variable slot for **Left Hand**
13. Run the scene. You should now be controlling the cubes on screen as you move your head and left arm!

www.it-ebooks.info

How it works...

The `ZigEngageSingleUser` component applies observed (Kinect input) positions and rotations for skeleton objects to associated game objects in the scene. In step 12, you linked the transforms of the cubes you created with particular skeleton elements that the `ZigEngageSingleUser` component is tracking form the Kinect. For each frame, this component applies the position and transform changes to your cubes, and we see our block character move as we move ourselves in front of the camera.

There's more...

Here are some details you don't want to miss.

Completing your block character

To complete your block character, you might use a capsule game object for the torso, and cubes for the other parts of the arms and legs. You might wish to refer to the Zigfu `Blockman3rdPerson` sample scene, since it is essentially the same as this recipe when it has been completed.

See also

- The *Control characters in Unity with the Microsoft Kinect using the Zigfu samples* recipe.
- The *Homemade mocap by storing movements from the Microsoft Kinect controller* recipe.

Homemade mocap by storing movements from the Microsoft Kinect controller

The positions and rotations of each skeleton component are available to access at any frame, therefore, if we create a data structure to store them, then a recording (and playback) of motions controlled via the Kinect can be implemented. This is demonstrated in the following recipe.

Getting ready

This recipe assumes you already have a project that uses Zigfu to control a character. You could either start with one of the scenes from the ZDK, or you could also build on the project from the previous recipe, where you created your own block character (that's what we did).

www.it-ebooks.info

How to do it...

1. Open your existing **Zigfu** project.
2. Create a sphere named `GhostHandL`, ensuring it is similar in size to the hand of the character in your scene (so this would be (0.2,0.2,0.2) if you are building on the previous recipe).
3. Add a red material to `GhostHandL`.

It doesn't matter what position `GhostHandL` starts at since it is going to follow the path of the left hand of your model once you start play back.

4. Create a new `ObjectAtFrame` C# script with the following code:

```
// file: ObjectAtFrame.cs
using UnityEngine;
using System.Collections;

public class ObjectAtFrame {
  public Vector3 position;
  public Quaternion rotation;

  public ObjectAtFrame(Vector3 newPosition, Quaternion
newRotation) {
    position = newPosition;
    rotation = newRotation;
  }
}
```

5. Add the following script class to the **Main Camera**:

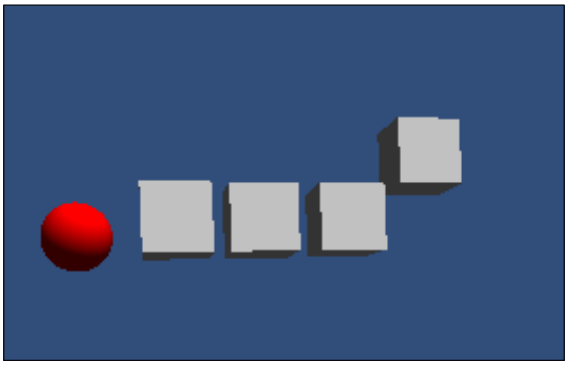
```
// file: RecordMovements.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class RecordMovements : MonoBehaviour {
  public Transform leftHand;
  public Transform redSphere;
  private bool isRecording = false;
  private List<ObjectAtFrame> movementList = new
List<ObjectAtFrame>();
  private int currentFrame = 0;

  private void OnGUI() {
```


www.it-ebooks.info

```
    if( !isRecording ) {
      bool startRecordingButtonClicked = GUILayout.Button ("START
recording");
      if( startRecordingButtonClicked ) {
         isRecording = true;
         movementList.Clear();
      }
    } else {
      GUILayout.Label ( "RECORDING --------------------" );
      GUILayout.Label("number of frames record = " + movementList.
Count);
      bool stopRecordingButtonClicked = GUILayout.Button ("STOP
recording");
      if( stopRecordingButtonClicked ){
        isRecording = false;
      }
    }
  }

  private void Update () {
    if( isRecording )
      StoreFrame();
    else
      if( movementList.Count > 0)
        PlayBackFrame();
  }

  private void StoreFrame() {
    ObjectAtFrame leftHandAtFrame = new ObjectAtFrame(leftHand.
position, leftHand.rotation);
    movementList.Add (leftHandAtFrame);
  }

  private void PlayBackFrame(){
    currentFrame++;
    if( currentFrame > (movementList.Count -1))
      currentFrame = 0;

    ObjectAtFrame objectNow = movementList[currentFrame] as
ObjectAtFrame;
    redSphere.position = objectNow.position;
    redSphere.rotation = objectNow.rotation;
  }
}
```


www.it-ebooks.info

6. With the **Main Camera** selected in the **Hierarchy** view, associate the following objects in the **Hierarchy** view to their corresponding public variables in the **Inspector**:
    - Drag **redSphere** into the **Red Sphere** variable
    - Drag the left hand of your character into the **Left Hand** variable



7. Run the project. Once the Kinect is controlling your character, click the **START recording** button, move your left hand in a big circle, and then click the **STOP recording** button.
8. You should then see the red sphere following your recorded left hand movement of a big circle (it will cycle through this recorded movement repeatedly).

How it works...

Important data structures for this recipe:

- The `ObjectAtFrame` class provides a data structure to store the position and rotation of one part of the character's body for a given frame.
- The `RecordMovements` class attached to the Main Camera, uses a generic `List` data structure named `movementList`, to create a dynamic ordered collection of `ObjectAtFrame` objects.

When the **START recording** button is clicked, the `isRecording` Boolean flag is set to `true`, and `movementList` is emptied with the `Clear()` method. The `Update()` method is executed each frame, and if `isRecording` is `true`, the `StoreFrame()` method is called.

The `StoreFrame()` method retrieves the position (`Vector3`) and rotation (`Quaternion`) of the **Transform** component of the left hand of the character (this was associated before running the project, by dragging that left hand object of the character to the public **Left Hand** variable (`leftHand`)). A new `ObjectAtFrame` object `leftHandAtFrame` is created with the position and rotation values, and added to the end of `movementList`.

www.it-ebooks.info

When the **STOP recording** button is clicked, the `isRecording` Boolean flag is set to `false`. The `Update()` method is executed for each frame, and if `isRecording` is `false`, and we have recorded `1` or more frames (that is, the `Count()` method of `movementList` is greater than zero), the `PlayBackFrame()` method is called.

The `PlayBackFrame()` method increments the `currentFame` integer, and resets it back to zero if its value is greater than the index of the last element in `movementList`. The position and rotation of the element for `currentFame` in `movementList` is retrieved, and applied to the **Transform** component of our `redSphere` object. Thus, the red sphere is made to follow the movement and rotation we recorded for that frame.

There's more...

Here are some details you don't want to miss.

Recording movements of multiple body parts

This recipe has illustrated the "proof of concept" of how to record motion movements of a single skeleton object in Unity with the Kinect device. However, it is more likely that we would want to record the position and rotation of *every* object in the skeleton. This would require a slightly more sophisticated data structure. If performance was not an issue, perhaps a `Dictionary` data structure could be used, with identifiers for each part of the skeleton. If efficiency were important, then perhaps parallel arrays, two for each skeleton object, could be implemented (one for position `Vector3` objects, and one for rotation `Quaternion` objects). Using a fixed number of array elements (that is, a maximum number of frames) would be the trade-off of speed against flexibility for this approach.

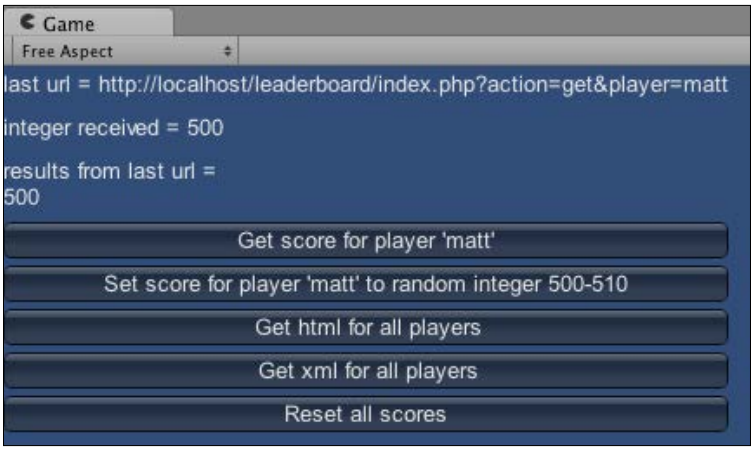
See also

- The *Control characters in Unity with the Microsoft Kinect using the Zigfu samples* recipe.
- The *Animating your own characters with the Microsoft Kinect controller* recipe.

Setting up a leaderboard using PHP/MySQL

Games are more fun when there is a leaderboard of high scores that players have achieved. Even single-player games can communicate to a shared web-based leaderboard. This recipe includes both the client-side (Unity) code, as well as the web server-side PHP scripts to set and get player scores from a MySQL database, as seen in the following screenshot:

www.it-ebooks.info



Getting ready

This recipe assumes you either have your own web hosting, or are running a local web server and database server such as XAMPP or MAMP. Your web server needs to support PHP, and you also need to be able to create MySQL databases.

All the SQL, PHP, and C# scripts for this recipe can be found in the `0423_07_10` folder.

If you are hosting your leaderboard on a public website, you should change the names of the database, database user, and password for security reasons. You should also implement some form of secret game code, as described in the *There's more...* section.

How to do it...

1. On your server, create a new MySQL database named `cookbook_highscores`.
2. On your server, create a new database user (`username=cookbook`. `password=cookbook`), with full rights to the database you just created.
3. On your server, execute the following SQL to create the `score_list` database table:

```
CREATE TABLE 'score_list' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'player' varchar(25) NOT NULL,
  'score' int(11) NOT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```


www.it-ebooks.info

4. On your server, copy these provided PHP script files to your web server:
   - index.php
   - scoreFunctions.php
   - htmlDefault.php
5. Add the following script class to the **Main Camera**:

```
// file: WebLeaderBoard.cs
using UnityEngine;
using System.Collections;
using System;

public class WebLeaderBoard : MonoBehaviour {
  private string url;
  private string action;
  private string parameters;
  private string textFileContents = "(still loading file ...)";

  private void OnGUI() {
    // hide closing tag
    string prettyText = textFileContents.Replace("</", "?@?");

    // prefix opening tag with newline
    prettyText = prettyText.Replace("<", "\n<");

    // return closing tag
    prettyText = prettyText.Replace("?@?", "</");

    GUILayout.Label ( "last url = " + url );
    GUILayout.Label ( StringToInt(textFileContents) );
    GUILayout.Label ( "results from last url = " + prettyText );

    WebButtons();
  }

  private void WebButtons() {
    bool getButtonWasClicked = GUILayout.Button("Get score for
player 'matt'");
    bool setButtonWasClicked = GUILayout.Button("Set score for
player 'matt' to random integer 500-510");
    bool htmlButtonWasClicked = GUILayout.Button("Get html for all
players");
    bool xmlButtonWasClicked = GUILayout.Button("Get xml for all
players");
```


www.it-ebooks.info

```
    bool resetButtonWasClicked = GUILayout.Button("Reset all
scores");

    if( getButtonWasClicked )
      GetAction();
    if( setButtonWasClicked )
      SetAction();
    if( htmlButtonWasClicked )
      HTMLAction();
    if( xmlButtonWasClicked )
      XMLAction();
    if( resetButtonWasClicked )
      ResetAction();
  }

  private string StringToInt(string s) {
    string intMessage = "integer received = ";
    try{
      int integerReturned = Int32.Parse(s);
      intMessage += integerReturned;
    }
    catch(System.Exception e){
      intMessage += "(not an integer) ";
      print (e);
    }
    return intMessage;
  }

  private void GetAction() {
    action = "get";
    parameters = "&player=matt";
    StartCoroutine( LoadWWW() );
  }

  private void SetAction() {
    int randomScore = UnityEngine.Random.Range(500, 510);
    parameters = "&player=matt&score=" + randomScore;
    action = "set";
    StartCoroutine( LoadWWW() );
  }

  private void HTMLAction() {
    action = "html";
    parameters = "";
```


www.it-ebooks.info

```
        StartCoroutine( LoadWWW() );
    }

    private void XMLAction() {
        action = "xml";
        parameters = "";
        StartCoroutine( LoadWWW() );
    }

    private void ResetAction() {
        action = "reset";
        parameters = "";
        StartCoroutine( LoadWWW() );
    }

    private IEnumerator LoadWWW(){
        string baseUrl = "http://localhost/leaderboard/index.
php?action=";
        url = baseUrl + action + parameters;
        WWW www = new WWW (url);
        yield return www;
            textFileContents = www.text;
    }
}
```

How it works...

The player's scores are stored in a MySQL database. Access to the basis is facilitated through the PHP scripts provided. In our example, all the PHP scripts were placed into a folder named `leaderboard` in the web server root folder. The scripts are accessed via the `http://localhost/leaderboard/` URL. All access is through the `index.php` PHP file. There are five actions implemented, and each is indicated by adding the action name at the end of the URL (this is the GET HTTP method, sometimes used for web forms—take a look at the address bar of your browser next time you search Google, for example). The actions and their parameters (if any) are as follows:

- `action` = `get`, parameters: `player` = `matt`:
  - This action asks for the integer score of the named player to be found
  - Returns: score integer
- `action` = `set`, parameters: `player` = `matt`, `score` = `101`:
  - This action asks for the provided score of the named player to be stored in the database (but only if this new score is greater than the currently stored score)

www.it-ebooks.info

    - Returns: score integer (if database update was successful), otherwise a negative value (to indicate no update took place)

- `action` = `html`, no parameters:
    - This action asks for HTML text listing all player scores to be returned
    - Returns: HTML text
- `action` = `xml`, no parameters:
    - This action asks for XML text listing all player scores to be returned
    - Returns: XML text
- `action` = `reset`, no parameters:
    - This action asks for a set of default player name and score values to replace the current contents of the database table
    - Returns: the `reset` string

The `OnGUI()` method first displays the current URL string, any integer value that was extracted from the response message received from the server, and the full text string received from the server. Also, five buttons are offered to the user, which set up the corresponding action and parameters to be added to the URL for the next call to the web server via the `LoadWWW()` method.

There's more...

Here is some information on how to fine-tune and customize this recipe.

Extract the full leaderboard data as XML for display within Unity

The XML text that can be retrieved from the PHP web server provides a useful method of allowing a Unity game to retrieve the full set of leaderboard data from the database. This allows the leaderboard to be displayed to the user in the Unity game (perhaps in some nice 3D fashion, or through a game-consistent GUI). See the next chapter for several recipes that illustrate ways to work with XML data in Unity.

Using secret game codes to secure your leaderboard scripts

The Unity and PHP code presented illustrates a simple, unsecured web-based leaderboard. To prevent players "hacking" into the board with false scores, it is common to encode some form of secret game code (or "key") into the communications. Only update requests that include the correct code will actually cause a change to the database.

www.it-ebooks.info

The Unity code would combine the secret key (in this example, the `harrypotter` string), with something related to the communication. For example, the same MySQL/PHP leaderboard may have different database records for different games, identified with a game ID:

```
// Unity Csharp code
string key = "harrypotter"
string gameId = 21;
string gameCode = Utility.Md5Sum(key + gameId);
```

The server-side PHP code would receive both the encrypted game code, and also the piece of game data used to create that encrypted code (in this example, the game ID, and the MD5 hashing function, which is available both in Unity and in PHP). The secret key (`harrypotter`) is used with the game ID to create an encrypted code that can be compared with the code received from the Unity game (or whatever user agent or browser is attempting to communicate with the leaderboard server scripts). Database actions will only be executed if the game code created on the server matches that sent along with the request for a database action:

```
// PHP - security code
$key = "harrypotter"
$game_id =  $_GET['game_id'];
$provided_game_code =  $_GET['game_code'];
$server_game_code = md5($key.$game_id);

if( $server_game_code == $provided_game_code ) {
  // codes match - do processing here
}
```

See also

- *Loading and parsing external XML files* in *Chapter 8*, *Working with External Text Files and XML Data*.
- *Preventing your game from running on unknown servers* in *Chapter 10*, *Improving Games with Extra Features and Optimization*.

www.it-ebooks.info

8

Working with External Text Files and XML Data

In this chapter, we will cover:

- Loading external text files using the TextAsset public variable
- Loading external text files using C# file streams
- Saving external text files with C# file streams
- Loading and parsing external XML files
- Creating XML text data manually using XMLWriter
- Creating XML text data automatically through serialization
- Creating XML text files – saving XML directly to text files with XMLDocument.Save()

Introduction

Text-based external data is very common and very useful as it is both computer and human readable. Text files may be used to allow non-technical team members to edit written content or for recording game performance data during development and testing. There is a lot of XML-based text file data on the Web, which is why we have included several XML-specific file reading and writing methods.

In the previous chapter, three general methods for loading external resource files were demonstrated, all of which work for text files as well as textures and audio files. Several additional methods for loading text files in particular are presented in this chapter.

All the recipes in this chapter are closely related.

www.it-ebooks.info

Loading external text files using the TextAsset public variable

A straightforward way to store data in text files and then choose between them before compiling is to use a public variable of the class `TextAsset`.

This technique is only appropriate when there will be no change to the data file after game compilation.

Getting ready

For this recipe, you'll need a text (`.txt`) file. In the `0423_08_01` folder, we have provided two such files:

- `cities.txt`
- `countries.txt`

How to do it...

To load external text files using `TextAsset`, perform the following steps:

1. Import the text file you wish to use into your project (for example, `cities.txt`)
2. Add the following C# script to **Main Camera**:

```
// file: ReadPublicTextAsset.cs
using UnityEngine;
using System.Collections;

public class ReadPublicTextAsset : MonoBehaviour {
    public TextAsset dataTextFile;
    private string textData = "";

    private void Start() {
        textData = dataTextFile.text;
    }

    private void OnGUI() {
        GUILayout.Label ( textData );
    }
}
```

3. With **Main Camera** selected in the **Hierarchy** view, drag the `cities.txt` file into the public string variable **dataTextFile** in the **Inspector** view.

www.it-ebooks.info

How it works...

The contents of the text file are read from the `dataTextFile TextAsset` object and stored as a string into `textData`. Our `OnGUI()` method displays the contents of `textData` as a label.

Loading external text files using C# file streams

For standalone executable games that both read from and write to (create or change) text files, .NET data streams are often used for both reading and writing. A later recipe illustrates how to write text data to files, while this recipe illustrates how to read a text file.

This technique only works when you compile to a Windows or Mac *standalone executable*; it will not work for Web Player builds, for example.

Getting ready

For this recipe, you'll need a text file; two of these have been provided in the `0423_08_01` folder.

How to do it...

To load external text files using C# file streams, perform the following steps:

1. Create a new C# script called `FileReadWriteManager`:

```
// file: FileReadWriteManager.cs
using System;
using System.IO;

public class FileReadWriteManager {
    public void WriteTextFile(string pathAndName, string
stringData) {
        // remove file (if exists)
        FileInfo textFile = new FileInfo(  pathAndName );
        if( textFile.Exists )
            textFile.Delete();

        // create new empty file
        StreamWriter writer;
        writer = textFile.CreateText();

        // write text to file
```


www.it-ebooks.info

```
            writer.Write(stringData);

            // close file
            writer.Close();
        }

        public string ReadTextFile(string pathAndName) {
            string dataAsString = "";

            try {
                // open text file
                StreamReader textReader = File.OpenText( pathAndName
);

                // read contents
                dataAsString = textReader.ReadToEnd();

                // close file
                textReader.Close();

            }
            catch (Exception e) {
                //display/set e.Message error message here if you wish

            }

            // return contents
            return dataAsString;
        }
}
```

2. Add the following C# script to **Main Camera**:

```
// file: ReadWithStream.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class ReadWithStream : MonoBehaviour {
    private string filePath = "";
    private string textFileContents = "(file not found yet)";
    private FileReadWriteManager fileReadWriteManager = new
FileReadWriteManager();

    private void Start () {
        string fileName = "cities.txt";
        filePath = Path.Combine(Application.dataPath,
"Resources");
```


www.it-ebooks.info

```
        filePath = Path.Combine(filePath, fileName);

        textFileContents = fileReadWriteManager.ReadTextFile(
filePath );
    }

    private void OnGUI() {
        GUILayout.Label ( filePath );
        GUILayout.Label ( textFileContents );
    }
}
```

3. Build your (Windows, Mac, or Linux) standalone executable. You'll need to save the current scene and then add this to the scenes in the build.
4. Copy the text file containing your data into your standalone's `Resources` folder (that is, the filename you set in the first statement in the `Start()` method—in our listing, this is the `cities.txt` file)

You will need to place the files in the `Resources` folder manually after every compilation.

For Windows and Linux users: When you create a Windows or Linux standalone executable, there is a `_Data` folder that is created with the executable application file. The `Resources` folder can be found inside this data folder.

For Mac users: A Mac standalone application executable looks like a single file, but it is actually a Mac OS "package" folder. Right-click on the executable file and select **Show Package Contents**. You will then find the standalone's `Resources` folder inside the `Contents` folder.

How it works...

When the game runs, the `Start()` method creates the `filePath` string and then calls the `ReadTextFile()` method from the `fileReadWriteManager` object, to which it passes it the `filePath` string. This method reads the contents of the file and returns them as a string, which is stored in the `textFileContents` variable. Our `OnGUI()` method displays the values of these two variables (`filePath` and `textFileContents`).

Note the need to use the `System.IO` package for this recipe. The C# script `FileReadWriteManager.cs` contains two general purpose file read and write methods that you may find useful in many different projects.

www.it-ebooks.info

Saving external text files with C# file streams

This recipe illustrates how to use C# streams to write text data to a text file, either into the standalone project's `Data` folder or to the `Resources` folder.

This technique only works when you compile to a Windows or Mac standalone executable.

Getting ready

In the `0423_08_02` folder, we have provided the `FileReadWriteManager.cs` class script file:

How to do it...

To save external text files using C# file streams, follow these steps:

1. Import the C# script `FileReadWriteManager.cs` into your project.
2. Add the following C# script to **Main Camera**:

```
// file: SaveTextFile.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class SaveTextFile : MonoBehaviour {
    public string fileName = "hello.txt";
    public string folderName = "Data";
    private string filePath = "(no file path yet)";
    private string message = "(trying to save data)";
    private FileReadWriteManager fileManager;

    void Start () {
        fileManager = new FileReadWriteManager();
        filePath = Application.dataPath + Path.
DirectorySeparatorChar + fileName;

        string textData = "hello \n and goodbye";
        fileManager.WriteTextFile( filePath, textData );

        message = "file should have been written now ...";
```


www.it-ebooks.info

```
    }

    void OnGUI()
    {
        GUILayout.Label("filepath = " + filePath);
        GUILayout.Label("message = " + message);
    }
}
```

3. Build and run your (Windows or Mac) standalone executable. You'll need to save the current scene and then add this to the scenes in the build.
4. You should now find a new text file named `hello.txt` in the `Data` folder of your project's standalone files, containing the lines "hello" and " and goodbye".

It is possible to test this when running within the *Unity editor* (that is, before building a standalone application). In order to test this way, you'll need to create a `Data` folder in your project panel.

How it works...

When the game runs, the `Start()` method creates the `filePath` string from the public variables `fileName` and `folderName`, and then calls the `WriteTextFile()` method from the `fileReadWriteManager` object, to which it passes the `filePath` and `textData` strings. This method creates (or overwrites) a text file (for the given file path and filename) containing the string data received.

There's more...

Some details you don't want to miss:

Choosing the Data or the Resources folder

Standalone build applications contain both a `Data` folder and a `Resources` folder. Either of these can be used for writing (or some other folder, if desired). We generally put read-only files into the `Resources` folder and use the `Data` folder for files that are to be created from scratch or have had their contents changed.

Loading and parsing external XML files

XML is a common data exchange format; it is useful to be able to parse (process the contents of) text files and strings containing data in this format. C# offers a range of classes and methods to make such processing straightforward.

www.it-ebooks.info

Getting ready

You'll find player name and score data in XML format in the `playerScoreData.xml` file in the `0423_08_04` folder. The contents of this file are as follows:

```
<scoreRecordList>
    <scoreRecord>
        <player>matt</player>
        <score>2200</score>
        <date>
            <day>1</day>
            <month>Sep</month>
            <year>2012</year>
        </date>
    </scoreRecord>

    <scoreRecord>
        <player>jane</player>
        <score>500</score>
        <date>
            <day>12</day>
            <month>May</month>
            <year>2012</year>
        </date>
    </scoreRecord>
</scoreRecordList>
```

The data is structured by a root element named `scoreRecordList`, which contains a sequence of `scoreRecord` elements. Each `scoreRecord` element contains a `player` element (which contains a player's name), a `score` element (which has the integer content of the player's score), and a `date` element, which itself contains three child elements, `day`, `month`, and `year`.

How to do it...

To load and parse external XML files, follow these steps:

1. Add the following C# script to **Main Camera**:

```
// file: ParseXML.cs
using UnityEngine;
using System.Collections;

using System.Xml;
using System.IO;
```

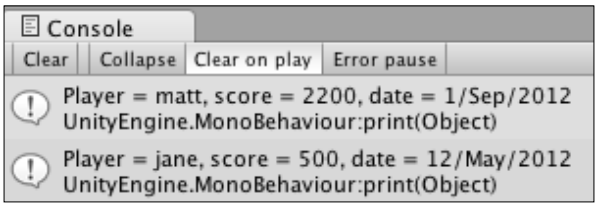

www.it-ebooks.info

```
public class ParseXML : MonoBehaviour {
    public TextAsset scoreDataTextFile;

    private void Start() {
        string textData = scoreDataTextFile.text;
        ParseScoreXML( textData );
    }

    private void ParseScoreXML(string xmlData) {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load( new StringReader(xmlData) );

        string xmlPathPattern = "//scoreRecordList/scoreRecord";
        XmlNodeList myNodeList = xmlDoc.SelectNodes(
xmlPathPattern );

        foreach(XmlNode node in myNodeList)
            print ( ScoreRecordString( node ) );

    }

    private string ScoreRecordString(XmlNode node) {
        XmlNode playerNode = node.FirstChild;
        XmlNode scoreNode = playerNode.NextSibling;
        XmlNode dateNode = scoreNode.NextSibling;

        return "Player = " + playerNode.InnerXml + ", score = " +
scoreNode.InnerXml + ", date = " + DateString( dateNode );
    }

    private string DateString(XmlNode dateNode) {
        XmlNode dayNode = dateNode.FirstChild;
        XmlNode monthNode = dayNode.NextSibling;
        XmlNode yearNode = monthNode.NextSibling;

        return dayNode.InnerXml + "/" + monthNode.InnerXml + "/" +
yearNode.InnerXml;
    }
}
```

2. Import the playerScoreData.xml file into your project.
3. Select the Main Camera and set playerScoreData.xml as the Score Data Text File variable for the Parse XML component.

www.it-ebooks.info

4. The output of the `print()` statements should be visible in the **Console** window:



How it works...

Note the need to use the `System.Xml` and `System.IO` packages for this recipe.

The `text` property of the `TextAsset` variable `scoreDataTextFile` provides the contents of the XML file as a string, which is passed to the `ParseScoreXML()` method. This method creates a new `XmlDocument` variable with the contents of this string. The `XmlDocument` class provides the `SelectNodes()` method, which returns a list of node objects for a given element path. In this example, a list of `scoreRecord` nodes is requested. A `for-each` statement prints out the string returned by the `ScoreRecordString()` method when each node object is passed to it.

The `ScoreRecordString()` method relies on the ordering of the XML elements to retrieve the player's name and score, and it gets the date as a slash-separated string by passing the node containing the three date components to the `DateString()` method.

There's more...

Some details you don't want to miss:

Retrieving XML data files from the web

You can use the `WWW` Unity class if the XML file is located on the Web rather than in your Unity project.

Creating XML text data manually using XMLWriter

One way to create XML data structures from game objects and properties is by hand-coding a method to create each element and its contents, using the `XMLWriter` class.

www.it-ebooks.info

How to do it...

To create XML text data using `XMLWriter`, follow these steps:

1. Add the following C# script to **Main Camera**:

```
// file: CreateXMLString.cs
using UnityEngine;
using System.Collections;
using System.Xml;
using System.IO;

public class CreateXMLString : MonoBehaviour {
    private string output = "(nothing yet)";

    private void Start () {
        output = BuildXMLString();
    }

    private void OnGUI() {
        GUILayout.Label( output );
    }

    private string BuildXMLString() {
        StringWriter str = new StringWriter();
        XmlTextWriter xml = new XmlTextWriter(str);

        // start doc and root el.
        xml.WriteStartDocument();
        xml.WriteStartElement("playerScoreList");

        // data element
        xml.WriteStartElement("player");
        xml.WriteElementString("name", "matt");
        xml.WriteElementString("score", "200");
        xml.WriteEndElement();

        // data element
        xml.WriteStartElement("player");
        xml.WriteElementString("name", "jane");
        xml.WriteElementString("score", "150");
        xml.WriteEndElement();

        // end root and document
        xml.WriteEndElement();
```


www.it-ebooks.info

```
            xml.WriteEndDocument();
            return str.ToString();
        }
    }
```

2. The XML text data should be visible when the game is run:

How it works...

The `Start()` method calls `BuildXMLString()` and stores the returned string in the `output` variable. Our `OnGUI()` method displays the contents of `output`.

The `BuildXMLString()` method creates a `StringWriter` object, into which `XMLWriter` builds the string of XML elements. The XML document starts and ends with the `WriteStartDocument()` and `WriteEndDocument()` methods. Elements start and end with `WriteStartElement(<elementName>)` and `WriteEndElement()`. String content for an element is added using `WriteElementString()`.

There's more...

Some details you don't want to miss:

Adding newlines to make XML strings more human readable

After every instance of the `WriteStartElement()` and `WriteElementString()` methods, you can add a newline character using `WriteWhiteSpace()`. These are ignored by XML parsing methods, but if you intend to display the XML string for a human to see, the presence of the newline characters makes it much more readable:

```
xml.WriteWhitespace("\n  ");
```

Making data class responsible for creating XML from list

The XML to be generated is often from a list of objects, all of the same class. In this case, it makes sense to make the class of the objects responsible for generating the XML for a list of those objects.

www.it-ebooks.info

The `CreateXMLFromArray` class simply creates an instance of `ArrayList` containing `PlayerScore` objects, and then calls the (static) method `ListToXML()`, passing in the list of objects.

```
// file: CreateXMLFromArray.cs
using UnityEngine;
using System.Collections;

public class CreateXMLFromArray : MonoBehaviour {
    private string output = "(nothing yet)";
    private ArrayList myPlayerList;

    private void Start () {
        myPlayerList = new ArrayList();
        myPlayerList.Add (new PlayerScore("matt", 200) );
        myPlayerList.Add (new PlayerScore("jane", 150) );

        output = PlayerScore.ListToXML( myPlayerList );
    }

    private void OnGUI() {
        GUILayout.Label( output );
    }
}
```

All the hard work is now the responsibility of the `PlayerScore` class. This class has two private variables for the players' name and score and a constructor that accepts values for these properties. The `public static` method `ListToXML()` takes an `ArrayList` object as an argument, and uses `XMLWriter` to build the XML string, looping through each object in the list and calling the object's `ObjectToElement()` method. This method adds an XML element to the `XMLWriter` argument received for the data in that object.

```
// file: PlayerScore.cs
using System.Collections;
using System.Xml;
using System.IO;

public class PlayerScore {
    private string _name;
    private int _score;

    public PlayerScore(string name, int score) {
        _name = name;
        _score = score;
    }
```


www.it-ebooks.info

```
        // class method
        static public string ListToXML(ArrayList playerList) {
            StringWriter str = new StringWriter();
            XmlTextWriter xml = new XmlTextWriter(str);

            // start doc and root el.
            xml.WriteStartDocument();
            xml.WriteStartElement("playerScoreList");

            // add elements for each object in list
            foreach (PlayerScore playerScoreObject in playerList) {
                playerScoreObject.ObjectToElement( xml );
            }

            // end root and document
            xml.WriteEndElement();
            xml.WriteEndDocument();

            return str.ToString();
        }

        private void ObjectToElement(XmlTextWriter xml) {
            // data element
            xml.WriteStartElement("player");
            xml.WriteElementString("name", _name);
            string scoreString = "" + _score; // make _score a string
            xml.WriteElementString("score", scoreString);
            xml.WriteEndElement();
        }
    }
```

Creating XML text data automatically through serialization

Another way to create XML data structures from game objects and properties is by serializing the contents of an object automatically. This technique automatically generates XML for all the public properties of an object.

Getting ready

In the `0423_08_06` folder, you'll find the listings for all four classes described in this recipe.

www.it-ebooks.info

How to do it...

To create XML text data through serialization, perform the following steps:

1. Create a C# script called `PlayerScore`:

```
// file: PlayerScore.cs
public class PlayerScore
{
    public string name;
    public int score;

    // default constructor, needed for serialization
    public PlayerScore() {}

    public PlayerScore(string newName, int newScore) {
        name = newName;
        score = newScore;
    }
}
```

2. Create a C# script called `SerializeManager`:

```
// file: SerializeManager.cs
//
// acknowledgements - this code has been adapted from:
//www.eggheadcafe.com/articles/system.xml.xmlserialization.asp
using System.Xml;
using System.Xml.Serialization;
using System.IO;
using System.Text;
using System.Collections.Generic;

public class SerializeManager<T> {
    public string SerializeObject(T pObject) {
        string XmlizedString = null;
        MemoryStream memoryStream = new MemoryStream();
        XmlSerializer xs = new XmlSerializer(typeof(T));
        XmlTextWriter xmlTextWriter = new
XmlTextWriter(memoryStream, Encoding.UTF8);
        xs.Serialize(xmlTextWriter, pObject);
        memoryStream = (MemoryStream)xmlTextWriter.BaseStream;
        XmlizedString = UTF8ByteArrayToString(memoryStream.
ToArray());
        return XmlizedString;
    }
```


www.it-ebooks.info

```
    public object DeserializeObject(string pXmlizedString) {
        XmlSerializer xs = new XmlSerializer(typeof(T));
        MemoryStream memoryStream = new MemoryStream(StringToUTF8B
yteArray(pXmlizedString));
        return xs.Deserialize(memoryStream);
    }

    private string UTF8ByteArrayToString(byte[] characters) {
        UTF8Encoding encoding = new UTF8Encoding();
        string constructedString = encoding.GetString(characters);
        return (constructedString);
    }

    private byte[] StringToUTF8ByteArray(string pXmlString) {
        UTF8Encoding encoding = new UTF8Encoding();
        byte[] byteArray = encoding.GetBytes(pXmlString);
        return byteArray;
    }
}
```

3. Add the following C# script class to **Main Camera**:

```
// file: SerialiseToXML.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SerialiseToXML : MonoBehaviour {
    private string output = "(nothing yet)";

    void Start () {
        SerializeManager<PlayerScore> serializer = new SerializeMa
nager<PlayerScore>();
        PlayerScore myData = new PlayerScore("matt", 200);
        output = serializer.SerializeObject(myData);
    }

    void OnGUI() {
        GUILayout.Label( output );
    }
}
```

4. Run your game. You should see XML data displayed as a label on the screen.

www.it-ebooks.info

How it works...

The `Start()` method creates `serializer`, a new `SerializeManager()` object for objects of the `PlayerScore` class. A new `PlayerScore` object, `myData`, is created with the values `"Matt"` and `200`. The `serializer` object is then passed the `myData` object, and the XML text data is returned as a string and stored in `output`. Our `OnGUI()` method displays the contents of `output`.

The `SerializeManager` class has been adapted from the code published by EggHeadCafe at the following URL:

`www.eggheadcafe.com/articles/system.xml.xmlserialization.asp`

You shouldn't need to worry too much about understanding all the code in this class—unless you want to :-). We have adapted the class so that it can be used for any data object class via C# generics; all you need to do is the following:

- Replace `<PlayerScore>` with `<YourDataClassName>` when declaring and creating an instance of `SerializationManager`; in our example, this is in the first statement of the `Start()` method
- Pass in an object of your class as the argument to `serializer.SerializeObject()`
- Ensure that your data class has a default constructor (that is public and takes no arguments)
- Make sure each property that you want included in the generated XML is public

The following site is a good place to learn more about Unity XML serialization:

`http://wiki.unity3d.com/index.php?title=Saving_and_Loading_Data:_XmlSerializer`

There's more...

Some details you don't want to miss:

Protecting member properties with accessor methods

Just having public properties is generally considered poor practice, because any part of an application with a reference to an object can make any kind of changes to those properties. However, if you wish to use this serialization recipe, you have to have public properties.

www.it-ebooks.info

A good solution is to provide public accessor methods, which behave like public properties but allow you to add a validation code behind the get/set methods for each corresponding hidden private property. Here, we have provided such an improved implementation of the `PlayerScore` class, `PlayerScore2`. Data is stored in the private variables `_name` and `_score`; they are accessed via the public variables `Name` and `Score`, which have get/set statements that handle changes to the private variables, and for which the appropriate validation logic could be implemented (for example, only changing `Score` if the new score is zero or higher, to prevent negative scores):

```
// file: PlayerScore2.cs
using System.Xml.Serialization;

[XmlRoot("player_score_2")]
public class PlayerScore2
{
    private string _name;
    private int _score;

    [XmlElement("name")]
    public string Name {
        get{ return _name; }
        set{ _name = value; }
    }

    [XmlElement("score")]
    public int Score {
        get{ return _score;}
        set{ _score = value;}
    }

    // default constructor, needed for serialization
    public PlayerScore2() {}

    public PlayerScore2(string newName, int newScore) {
        Name = newName;
        Score = newScore;
    }
}
```

Variable naming styles vary between programmers and organizations. One approach to naming private member variables is to prefix them with an underscore (as we have done in the previous code snippet). An alternative, which some programmers prefer, involves using no underscore and disambiguating the argument of a setter method from the member variable with the same identifier by using keyword `this`; for example, `this.name = name`.

www.it-ebooks.info

Creating XML text files – saving XML directly to text files with XMLDocument.Save()

It is possible to create an XML data structure and then save that data directly to a text file using the `XMLDocument.Save()` method; this recipe illustrates how.

How to do it...

To save XML data to text files directly, perform the following steps:

1. Create a new C# script named `PlayerXMLWriter`:

```
// file: PlayerXMLWriter.cs
using System.Text;
using System.Xml;
using System.IO;

public class PlayerXMLWriter {
    private string _filePath;
    private XmlDocument _xmlDoc;
    private XmlElement _elRoot;

    public PlayerXMLWriter(string filePath) {
        _filePath = filePath;
        _xmlDoc = new XmlDocument();

        if(File.Exists (_filePath)) {
            _xmlDoc.Load(_filePath);
            _elRoot = _xmlDoc.DocumentElement;
            _elRoot.RemoveAll();
        }
        else {
            _elRoot = _xmlDoc.CreateElement("playerScoreList");
            _xmlDoc.AppendChild(_elRoot);
        }
    }

    public void SaveXMLFile() {
        _xmlDoc.Save(_filePath);
    }
    public void AddXMLElement(string playerName, string
playerScore) {
        XmlElement elPlayer = _xmlDoc.
CreateElement("playerScore");
```


www.it-ebooks.info

```
            _elRoot.AppendChild(elPlayer);

            XmlElement elName = _xmlDoc.CreateElement("name");
            elName.InnerText = playerName;
            elPlayer.AppendChild(elName);

            XmlElement elScore = _xmlDoc.CreateElement("score");
            elScore.InnerText = playerScore;
            elPlayer.AppendChild(elScore);
        }
}
```

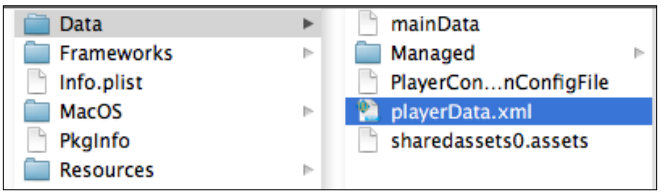
2. Add the following C# script to **Main Camera**:

```
// file: CreateXMLTextFile.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class CreateXMLTextFile : MonoBehaviour {
    public string fileName = "playerData.xml";
    public string folderName = "Data";

    private void Start() {
        string filePath = Application.dataPath + Path.
DirectorySeparatorChar + fileName;

        PlayerXMLWriter myPlayerXMLWriter = new
PlayerXMLWriter(filePath);
        myPlayerXMLWriter.AddXMLElement("matt", "55");
        myPlayerXMLWriter.AddXMLElement("jane", "99");
        myPlayerXMLWriter.AddXMLElement("fred", "101");
        myPlayerXMLWriter.SaveXMLFile();

        print( "XML file should now have been created at: " +
filePath);
    }
}
```

3. Build and run your (Windows, Mac, or Linux) standalone executable. You'll need to save the current scene and then add this to the list of scenes in the build.

www.it-ebooks.info

4. You should now find a new text file named `playerData.xml` in the `Data` folder of your project's standalone files, containing the XML data for the three players:



5. The contents of the `playerData.xml` file should be the XML player list data:

```
<playerScoreList>
  <playerScore>
    <name>matt</name>
    <score>55</score>
  </playerScore>
  <playerScore>
    <name>jane</name>
    <score>99</score>
  </playerScore>
  <playerScore>
    <name>fred</name>
    <score>101</score>
  </playerScore>
</playerScoreList>
```

How it works...

The `Start()` method creates `myPlayerXMLWriter`, a new object of the `PlayerXMLWriter` class, to which it passes the new, required XML text file `filePath` as an argument. Three elements are added to the `PlayerXMLWriter` object, which store the names and scores of three players. The `SaveXMLFile()` method is called and a debug `print()` message is displayed.

The `PlayerXMLWriter` class works as follows: when a new object is created, the provided file path string is stored in a private variable; at the same time, a check is made to see whether any file already exists. If an existing file is found, the content elements are removed; if no existing file is found, then a new root element, `playerScoreList`, is created as the parent for child data nodes. The `AddXMLElement()` method appends a new data node for the provided player name and score. The `SaveXMLFile()` method saves the XML data structure as a text file for the stored file path string.

www.it-ebooks.info

www.it-ebooks.info

9

Managing Object States and Controlling Their Movements

In this chapter, we will cover the following:

- Controlling cube movement with player controls
- Controlling object look-at behavior
- Controlling object-to-object movements (seek, flee, follow at a distance)
- Controlling object group movement through flocking
- Firing objects by instantiation with forward velocity
- Finding a random spawn point
- Finding the nearest spawn point
- Following waypoints in a sequence
- Managing object behavior with states
- Managing complex object behavior with the state pattern

www.it-ebooks.info

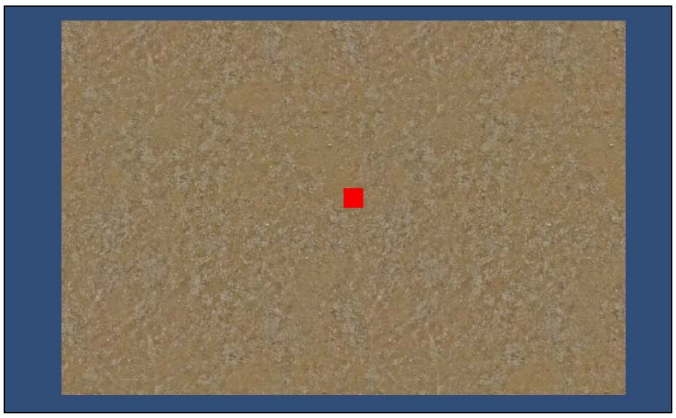
Introduction

Many games involve moving computer-controlled objects and characters. For some games, animation components can be sufficient for all such movements. However, other games use **directional logic**—simple or sophisticated artificial intelligence involving the computer program that makes objects orient and move towards or away from the player or other game objects and characters. This chapter presents a range of such directional recipes, from which many games can benefit in terms of a richer and more exciting user experience.

Unity provides sophisticated classes and components, including the `Vector3` class, and rigid body physics for modeling realistic movements, forces, and collisions in games. We make use of these game engine features to implement some sophisticated **non-player character** (**NPC**) object movements in the recipes of this chapter.

Controlling cube movement through player controls

Many of the recipes in this chapter are built on this basic project, which constructs a scene with a textured terrain, a downward facing **Main Camera**, and a red cube (as shown in the following screenshot) that can be moved around by the user using the four directional arrow keys.



How to do it...

To create a player-controlled cube, perform the following steps:

1. Start a new project by navigating to **Terrain Textures** | **GoodDirt.psd** from the **Terrain Assets** Unity package (uncheck all of the other contents of this package as we don't need them and they'll just bloat the size of the project).
2. Create a terrain, positioned at (-15, 0, -10) and sized 30 by 20.

www.it-ebooks.info

Transform position for terrains relates to their corner not their center

Since the `Transform` position of terrains relates to the corner of the object, we center such objects at (0, 0, 0) by setting the X-coordinate equal to (-1 x width/2) and the Z-coordinate to (-1 x length/2). In other words, we slide the object by half its width and half its height to ensure its center is just where we want it.

In this case, width is 30 and length is 20, hence we get -15 for X (-1 x 30/2), and -10 for Z (-1 x 20/2).

3. Set the material for this terrain to be **GoodDirt**.
4. Create a directional light (it should face downwards onto the terrain with default settings. If it doesn't for some reason, rotate it so the terrain is well lit).
5. Make the following changes to **Main Camera**:
   - **Position**: (0, 20, 0)
   - **Rotation**: (90, 0, 0)
6. Change the **Aspect Ratio** of the **Game Panel** from **Free Aspect** to **4:3**. You should now see the whole of the terrain in the **Game Panel**.
7. Create a new cube GameObject named `Cube-player`, at position (0, 0.5, 0) and sized (1, 1, 1).
8. Create a new red material named `m_red`, and apply this material to `Cube-player`.
9. Add the following C# script class to `Cube-player`:

```
// file: PlayerControl
using UnityEngine;
using System.Collections;

public class PlayerControl : MonoBehaviour {
  public float y;
  public const float MIN_X = -15;
  public const float MAX_X = 15;
  public const float MIN_Z = -10;
  public const float MAX_Z = 10;

  private float speed = 20;

  private void Awake(){
    y = transform.position.y;
  }
```


www.it-ebooks.info

```
  private void Update () {
    KeyboardMovement();
    CheckBounds();
  }

  private void KeyboardMovement(){
    float dx = Input.GetAxis("Horizontal") * speed * Time.
deltaTime;
    float dz = Input.GetAxis("Vertical") * speed * Time.deltaTime;
    transform.Translate( new Vector3(dx,y,dz) );
  }

  private void CheckBounds(){
    float x = transform.position.x;
    float z = transform.position.z;
    x = Mathf.Clamp(x, MIN_X, MAX_X);
    z = Mathf.Clamp(z, MIN_Z, MAX_Z);
    transform.position = new Vector3(x, y, z);
  }
}
```

10. Add the following C# script class to your project:

```
// file: UsefulFunctions.cs
using UnityEngine;

public class UsefulFunctions {
  public static void DebugRay(Vector3 origin, Vector3 v, Color c)
{
    Debug.DrawRay(origin, v * v.magnitude, c);
  }

  public static Vector3 ClampMagnitude(Vector3 v, float max) {
    if (v.magnitude > max)
      return v.normalized * max;
    else
      return v;
  }
}
```

How it works...

The scene contains a terrain, positioned so its center is (0, 0, 0). The red cube is controlled by the user's arrow keys through the `PlayerControl` script. When the `Awake()` method is called, the starting Y-coordinate is stored, so that any changes to the position of the cube will always keep the object to the same Y-position. For each frame, the `Update()` method first calls `KeyboardMovement()` and then `CheckBounds()`.

www.it-ebooks.info

The `KeyboardMovement()` method reads the horizontal and vertical input values (which the Unity default settings read from the four directional arrow keys). Based on these left-right and up-down values, the position of the player's cube is moved (translated). The amount it is moved depends on the speed variable. Since our camera is looking down, the Y-coordinate is fixed, and it is the X and Z values that are used to determine how to move the red cube.

The `CheckBounds()` method simply checks the X and Z positions against the four constants for the maximum and minimum values. If values are outside the range, they are set to the maximum or minimum as appropriate.

Mathf.Clamp() is a very handy function

This function is very useful as it allows us to restrict a value between a minimum and maximum value. The first parameter is the value, the second is the minimum, and the third is the maximum. If the value is smaller than the minimum, then the minimum is returned. If the value is larger than the maximum, the maximum is returned. Otherwise the value (which must then be somewhere within these limits) is returned.

Although not used in this project, the class `UsefulFunctions` is used by several of the recipes in this chapter. It offers two useful functions to our project:

- `DebugRay()`: This takes two `Vector3` objects and a color as input, and draws a line of the given color from the start point (first argument) in the direction (and length) of the vector, which is the second argument
- `ClampVector()`: This takes a `Vector3` object and a maximum size as input, and returns a vector with a maximum length (magnitude) of the input maximum size, that is, if the length of the given vector is greater than the maximum, then a vector of maximum length in the same direction is returned

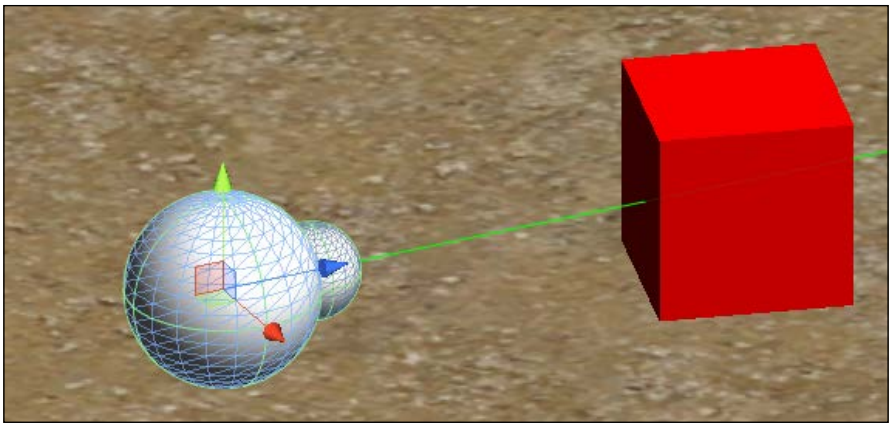
See also

- The Controlling object look-at behavior recipeThe Controlling object-to-object movements (seek, flee, follow at a distance) recipe
- The Controlling object group movement through flocking recipe
- The Finding a random spawn point recipe
- The Finding the nearest spawn point recipe
- The Following waypoints in a sequence recipe

www.it-ebooks.info

Controlling object look-at behavior

There are often situations where we want to make one object orient itself to look in the direction of some other object. Examples include enemies or gun turrets wanting to look at the player, or perhaps making the player start off looking at an object at a key point in the game, such as, after having been teleported or respawned to some location in the game.



Getting ready

This recipe builds upon the player-controlled cube Unity project you created in the previous recipe. So make a copy of that project, open it, and then follow the steps for this recipe.

How to do it...

To make one object "look-at" another object, perform the following steps:

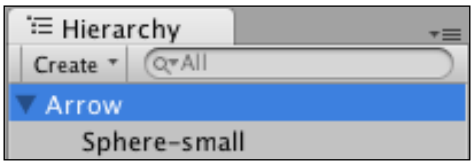
1. Create a sphere named `Arrow` positioned at (2, 0.5, 2) and with scale (1, 1, 1).
2. Create a second sphere, named `Sphere-small`, positioned at (2, 0.5, 2.5) and with scale (0.5, 0.5, 0.5).

By default, when objects are first created, they are "looking" (facing) along the Z-axis. By positioning `Sphere-small` at (0, 0, 0.5), we have made this smaller sphere the point of an arrow. In other words, when we look at the two spheres, they are facing in the direction of a line drawn from the center of the large sphere towards the center of the small sphere.

You can check whether this is correct in the **Scene** window. Selecting **Arrow**, you should see the blue Z-axis gizmo arrow point from the center of the larger sphere towards the center of the smaller sphere.

www.it-ebooks.info

3. Make `Sphere-small` a child-object of your GameObject `Arrow`.



To make an object a child of another object, in the **Hierarchy** window drag the child object into the parent object. The child object should then appear indented below the parent object. The parent object will also have a *contents* triangle symbol next to it, to control whether its contents (child objects) are listed or not.

4. Add the following C# script class to your GameObject `Arrow`:

```
// file: LookAt.cs
using UnityEngine;
using System.Collections;

public class LookAt : MonoBehaviour {
  public Transform playerTransform;

  private void LateUpdate () {
    transform.LookAt( playerTransform );


    UsefulFunctions.DebugRay(transform.position, transform.forward
* 100, Color.green);

  }
}
```

We use `LateUpdate()` so we can be sure `playerTransform` has been moved before calling `LookAt()`.

5. Ensuring **Arrow** is selected, in the **Inspector** for the `LookAt` scripted component, drag **Cube-player** over the public variable **playerTransform**.

www.it-ebooks.info

How it works...

Since it is so useful, Unity has provided all `Transform` objects with the `LookAt()` method. This method will make the object whose transform has the method applied, orient itself at the object whose transform is provided as a parameter. Since our `Arrow` object's script has a reference to `Cube-player` (via the public variable `playerTransform`), every frame the `Arrow` object is made to re-orient itself to, faces towards the direction of the player's character.

The second statement in the `LateUpdate()` method is the call to the `DebugRay()` method of the `UsefulFunctions` class. This illustrates that the "rays" (lines in 3D space) that can be drawn in the **Scene** window, and can aid in debugging. This statement will result in a ray being drawn in green, from the position of the Arrow object's transform, 100 Unity units in the direction the Arrow object is facing (its transform's `forward` vector).

See also

- The *Controlling cube movement through player controls* recipe
- The *Controlling object-to-object movements (seek, flee, follow at a distance)* recipe
- The *Controlling object group movement through flocking* recipe

Controlling object-to-object movements (seek, flee, follow at a distance)

Computer controlled characters (non-player characters) often need to seek and move towards another object (such as the player's character), or alternatively flee away from another character. The logic is the same for both seeking and fleeing, and is demonstrated in this recipe.

Getting ready

This recipe builds upon the player-controlled cube Unity project you created in the first recipe. So make a copy of that project, open it, and then follow the steps of this recipe.

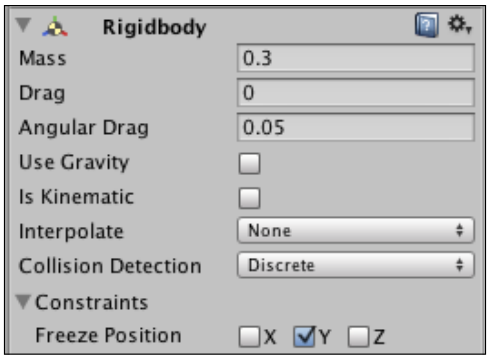
How to do it...

To make an object seek, flee, or follow another object, perform the following steps:

1. Create a sphere named `Arrow`, positioned at (2, 0.5, 2) and with scale (1, 1, 1).
2. Create a second sphere, named `Sphere-small`, positioned at (2, 0.5, 2.5) and with scale (0.5, 0.5, 0.5).
3. Make `Sphere-small` a child-object of your GameObject `Arrow`.

www.it-ebooks.info

4. Add a **Rigidbody** component from **Physics** to the GameObject `Arrow`, modifying the following properties:
   - Set the value of **Mass** to **0.3**
   - Uncheck the **Use Gravity** checkbox
   - Check (tick) the **Freeze Position** for the **Y** axis only



5. Add the following C# script class to your GameObject `Arrow`:

```
// file: SeekTarget.cs
using UnityEngine;
using System.Collections;

public class SeekTarget : MonoBehaviour {
  public GameObject playerGO;
  public const float MAX_MOVE_DISTANCE = 500.0f;

  private void FixedUpdate () {
    float moveDistance = MAX_MOVE_DISTANCE * Time.deltaTime;
    Vector3 source = transform.position;
    Vector3 target = playerGO.transform.position;

    Vector3 seekVelocity = Seek(source, target, moveDistance);
    seekVelocity = UsefulFunctions.ClampMagnitude(seekVelocity,
moveDistance);
    rigidbody.AddForce( seekVelocity, ForceMode.VelocityChange );
    UsefulFunctions.DebugRay(transform.position, seekVelocity,
Color.blue);
    UsefulFunctions.DebugRay(transform.position, rigidbody.
velocity, Color.yellow);
  }

  private Vector3 Seek(Vector3 source, Vector3 target, float
moveDistance){
```


www.it-ebooks.info

```
        Vector3 directionToTarget = Vector3.Normalize( target - source
);
        Vector3 velocityToTarget = moveDistance * directionToTarget;
        transform.LookAt( playerGO.transform );

        return velocityToTarget - rigidbody.velocity;
    }
}
```

6. Ensuring **Arrow** is selected, in the **Inspector** for the `SeekTarget` scripted component, drag **Cube-player** over the public variable **playerTransform**.

How it works...

We have added a Unity **Rigidbody** component from **Physics** to our `Arrow` object. This means we can ask Unity to apply a "force" to this object, and Unity will work out how to move the object based on its mass and any previous velocity the object already had.

When working with physics and forces, we put our logic into the `FixedUpdate()` method, which is called at a fixed frame rate, immediately before Unity runs its physics simulation for the frame. Had such logic been located in the `Update()` method, several `Update()` messages, or none, might have been received since the last physics' simulation update, leading to uneven behavior.

Remember, once you have added a physics rigid body to an object you are influencing through code, that code must be placed into the `FixedUpdate()` method.

One danger when adding forces to already moving objects is that they start to move very fast, and this can lead to "overshooting" the target, and sometimes can even leave the chasing object forever orbiting around the target, never getting any closer to it. Therefore, we define a `MAX_MOVE_DISTANCE` constant, which will set a limit on the size of any force we will apply to `Arrow`. As usual, since the frame rate of games vary, we need to use the proportion of this speed maximum that is relative to the time since the last frame (`Time.deltaTime`).

Each time `FixedUpdate()` is called, the maximum move distance for this frame (`moveDistance`) is calculated first. Then the `Seek()` method is called passing the current positions of the source (`Arrow`) and target (`Cube-player`) objects. This method returns a `Vector3` object that represents the force we wish to apply to `Arrow` to make it move towards the players cube. This force is limited to the size of `moveDistance`, and then applied to the `Arrow` object as a force to the rigid body component. Finally, the `Arrow` object is made to face towards its target, and then two debug rays are drawn, a blue one to show the seeking force we have just applied, and a yellow one to show the current direction/speed velocity that the `Arrow` object now has.

www.it-ebooks.info

The core of this target-seeking behavior is to make use of the following two key pieces of information, which we do in the `Seek()` method:

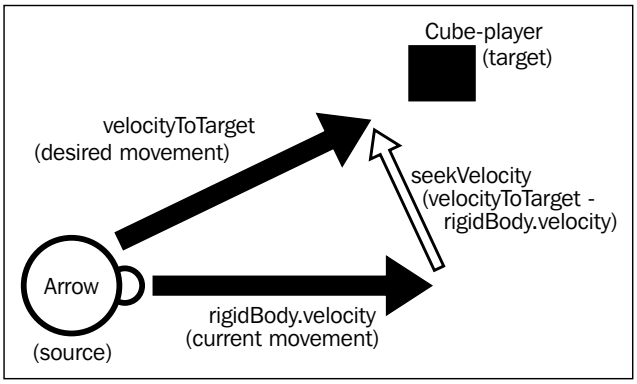
- The *direction* from the `Arrow` (source) object towards the player's cube (target) object
- The *magnitude* (length) of the force vector we want to apply to our `Arrow` object

A `Vector3` object is the perfect data structure for forces, since they represent both direction and magnitude (the length of the vector).

A **normalized** vector is simply a vector with a length (magnitude) of 1. This is very useful since we can then multiple a normalized vector by the speed (distance) we want an object to move. Sometimes we refer to a normalized vector as a **unit vector**, since it has a length of 1.

The `Seek()` method does the following:

- It calculates the direction from the `Arrow` (source) object towards the players cube (target) object by subtracting the source position from the target position, and then normalizing the result.
- This direction is multiplied by `moveDistance` to give the velocity vector we'd like the `Arrow` object to move.
- If `Arrow` is already moving, then simply applying the force towards the target may "nudge" the `Arrow` object a little off its current direction towards the target. What we really want is to calculate the force to apply to make `Arrow` move towards the target taking into account the `Arrow` object's current movement. We do this by subtracting the `Arrow` object's current movement (`rigidbody.velocity`) from the force in the direction of the target. The result is a force that will redirect `Arrow` from its current direction towards the target.



www.it-ebooks.info

There's more...

The following are some details you don't want to miss.

Using the simpler (and less sophisticated) MoveTowards() method

If the only movement behavior you wish to implement for an object is a simple seek behavior, Unity does provide a very straightforward method for this purpose. The use of rigid bodies and forces is needed for the later recipes in this chapter, which is why this simple method approach was not taken for this recipe. However, you could replace the code in this recipe for the `SeekTarget` class with the following, which uses the `MoveTowards()` Unity method for a simple seek behavior:

```
// file: MoveTowards.cs
using UnityEngine;
using System.Collections;

public class MoveTowards : MonoBehaviour {
  public Transform playerTransform;
  public float speed = 5.0F;

  private void Update () {
    transform.LookAt(playerTransform);
    float distance = speed * Time.deltaTime;
    Vector3 source = transform.position;
    Vector3 target = playerTransform.position;
    transform.position = Vector3.MoveTowards(source, target,
distance);
  }
}
```

The `MoveTowards()` method takes three arguments, namely: the source position, the target position, and how far to move towards the target.

Improving arrival behavior through deceleration when near target

A nicer version of the seek behavior involves the seeking object to slow down as it gets closer to the target object. This can be achieved by replacing the `Seek()` method with a new `Arrive()` method. First, replace the statement in `FixedUpdate()`, so that the value of `adjustVelocity` is returned from the `Arrive()` method:

```
Vector3 adjustVelocity = Arrive(source, target);
```


www.it-ebooks.info

Next, define a constant `DECELERATION_FACTOR` that we need:

```
const float DECELERATION_FACTOR = 0.6f;
```

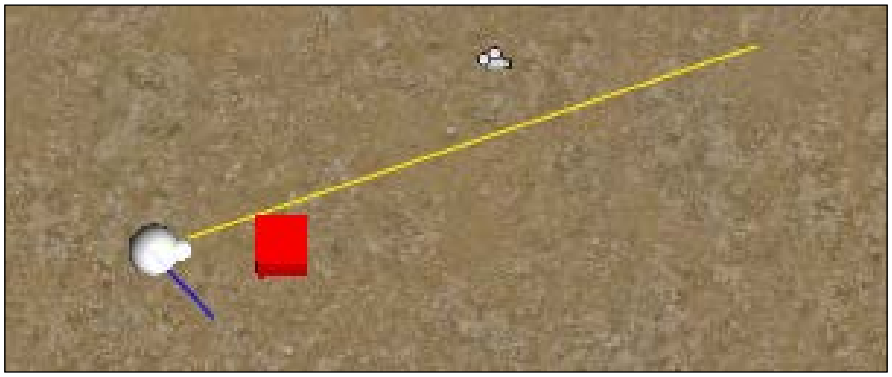
Finally, implement the `Arrive()` method with the following code:

```
private Vector3 Arrive(Vector3 source, Vector3 target){float
distanceToTarget = Vector3.Distance(source, target);
  Vector3 directionToTarget = Vector3.Normalize( target - source );
  float speed = distanceToTarget / DECELERATION_FACTOR;
  Vector3 velocityToTarget = speed * directionToTarget;

  return velocityToTarget - rigidbody.velocity;
}
```

The speed (magnitude) of the force to be applied to the `Arrow` object will vary depending on the remaining distance to the target (player's cube). This rate of deceleration can be tweaked by changing the value of the `DECELERATION_FACTOR` constant. As the distance to the target is reduced (that is, as the `Arrow` object gets closer), the size of the force to be applied to `Arrow` is smaller, so it slows down as it gets closer to the target. This results in a more natural (and satisfying) movement in target-seeking game objects.

The next screenshot shows the following two debug rays of the forces on the `Arrow` object at a point in time:

- The longer ray shows the current movement velocity of the Arrow—upwards and to the right
- The shorter ray shows the force to be applied to `Arrow` to make it move towards the player's cube, downwards to the right—which should result in the `Arrow` colliding with the target player cube object very soon



www.it-ebooks.info

Implementing flee target and follow at a distance

The opposite to arrive at is the flee behavior. This involves moving in the opposite direction to where the target is. First, change `FixedUpdate()` so that the value of `adjustVelocity` is set differently according to the distance from target (its default is `0`; if within the value of `FLEE_RADIUS`, it will be a force to move away from the target, if outside the value of `SEEK_RADIUS`, it will be a force to move towards the target):

```
Vector3 adjustVelocity = Vector3.zero;

float distanceToTarget = Vector3.Distance(target, source);
if( distanceToTarget < FLEE_RADIUS )
  adjustVelocity += Flee(source, target);

if( distanceToTarget > SEEK_RADIUS )
  adjustVelocity += Arrive(source, target);
```

Next, define some constants we need:

```
const float DECELERATION_FACTOR = 0.6f;
public const float FLEE_RADIUS = 14f;
public const float SEEK_RADIUS = 6f;
```

As the value of `FLEE_RADIUS` is larger than that of `SEEK_RADIUS`, when the distance to the target is between 6 and 14, then *both* a flee and an arrive force will be calculated, resulting in a deceleration in this overlap zone. The result is a realistic following behavior, whereby the Arrow follows at approximately 10 units, accelerating and decelerating smoothly based on the player's cube movements.

Add the `Arrive()` method listed in the previous section, and also add a new method called `Flee()` as follows:

```
private Vector3 Flee(Vector3 source, Vector3 target){
  float distanceToTarget = Vector3.Distance(target, source);
  Vector3 directionAwayFromTarget = Vector3.Normalize(source -
target);
  float speed = (FLEE_RADIUS - distanceToTarget) / DECELERATION_
FACTOR;
  Vector3 velocityAwayFromTarget = speed * directionAwayFromTarget;
  return velocityAwayFromTarget;
}
```

The `Flee()` method works in the opposite way to `Arrive()`, so the speed (magnitude) of the velocity force vector will be larger when the target is closer. Therefore, the `Arrow` will move quickly away from a nearby target.

A follow-at-a-distance behavior can be achieved by making an object flee a nearby target, but seek to arrive at a far away target.

www.it-ebooks.info

See also

- The *Controlling cube movement through player controls* recipe
- The *Controlling object look-at behavior* recipe
- The *Controlling object group movement through flocking* recipe
- The *Following waypoints in a sequence* recipe

Controlling object group movement through flocking

Realistic, natural looking flocking behavior (for example, birds, antelopes, or bats) can be created through creating collections of objects with the following four simple rules:

- **Separation**: Avoid getting too close to neighbors
- **Avoid obstacles**: Turn away from an obstacle which is immediately ahead
- **Alignment**: Move in the direction the flock is heading
- **Cohesion**: Move towards the location in the middle of the flock

Each member of the "flock" acts independently, but needs to know about the current heading and location of the members of its flock. This recipe shows you how to create a scene of flocking cubes. To keep things simple, we'll only implement basic versions of the last two rules of the preceding list: alignment and cohesion.

Getting ready

This recipe builds upon the player-controlled cube Unity project you created in the first recipe. So make a copy of that project, open it, and then follow the steps of this recipe.

How to do it...

To make a group of objects flock together, perform the following steps:

1. Add the following C# script class to the **Main Camera**:

```
// file: Swarm
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Swarm : MonoBehaviour {
  public int droneCount = 20;
```


www.it-ebooks.info

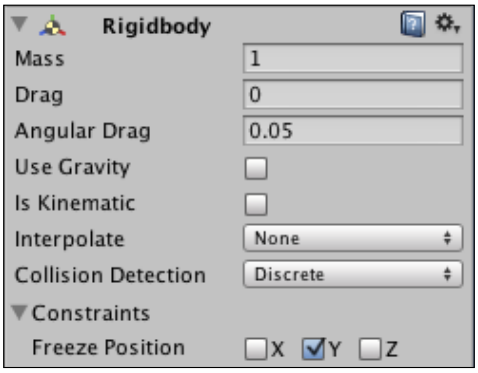
```
public GameObject dronePrefab;

private List<Drone> drones = new List<Drone>();

private void Awake(){
  for (int i = 0; i < droneCount; i++){
    AddDrone();
  }
}

private void AddDrone(){
  GameObject newDroneGO = (GameObject)Instantiate(dronePrefab);
  Drone newDrone = newDroneGO.GetComponent<Drone>();
  drones.Add( newDrone );

  float speed = 5f;
  float maxSpeed = 10f;
  float maxDirectionChange = .05f;
  newDrone.SetParameters(speed, maxSpeed, maxDirectionChange);
}

private void FixedUpdate(){
  Vector3 swarmCenter = SwarmCenterAverage();
  Vector3 swarmMovement = SwarmMovementAverage();

  foreach(Drone drone in drones ) {
    drone.UpdateVelocity(swarmCenter, swarmMovement);

  }
}

private Vector3 SwarmCenterAverage() {
  // cohesion (swarm center point)
  Vector3 locationTotal = Vector3.zero;

  foreach(Drone drone in drones ) {
    locationTotal += drone.transform.position;
  }

  return (locationTotal / drones.Count);
}

private Vector3 SwarmMovementAverage() {
  // alignment (swarm direction average)
```


www.it-ebooks.info

```
        Vector3 velocityTotal = Vector3.zero;

        foreach(Drone drone in drones ) {
          velocityTotal += drone.rigidbody.velocity;
        }

        return (velocityTotal / drones.Count);
      }
    }
```

2. Create a new cube GameObject named `Cube-boid`, at (0, 0, 0) and add a **RigidBody** component from **Physics** to `Cube-boid` with the following properties:
   - **Mass** as **1**
   - **Drag** is **0**
   - **Angular Drag** is **0.05**
   - **Use Gravity** and **Is Kinematic** are both un-checked
   - Under **Constrains Freeze Position** for the **Y** axis is checked

3. You should see the following Inspector values for your cube's rigid body component:



4. Add the following C# script class to `Cube-boid`:

```
// file: Drone.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Drone : MonoBehaviour {
  public void SetParameters(float speed, float maxSpeed, float
maxDirectionChange){
    _speed = speed;
```


www.it-ebooks.info

```
    _maxSpeed = maxSpeed;
    _maxDirectionChange = maxDirectionChange;
  }

  private float _speed = 5f;
  private float _maxSpeed = 10f;
  private float _maxDirectionChange = .05f;

  public void UpdateVelocity(Vector3 swarmCenterAverage, Vector3
swarmMovementAverage){
    // calc velocity adjustment for swarm
    Vector3 moveTowardsSwarmCenter = VectorTowards( swarmCenterAverage
);
    Vector3 adjustment = moveTowardsSwarmCenter + (2 *
swarmMovementAverage );
    Vector3 swarmVelocityAdjustment = UsefulFunctions.
ClampMagnitude(adjustment, _maxDirectionChange);

    // apply velocity adjustment to this drone
    rigidbody.velocity += (swarmVelocityAdjustment * _speed);
    rigidbody.velocity = UsefulFunctions.ClampMagnitude(rigidbody.
velocity, _maxSpeed);
  }

  private Vector3 VectorTowards(Vector3 target) {
    Vector3 targetDirection = target - transform.position;
    targetDirection.Normalize();
    targetDirection *= _maxSpeed;
    return (targetDirection - rigidbody.velocity);
  }
}
```

5. Create a new empty Prefab named `prefab_Boid`, and from the **Hierarchy** panel drag **Cube-boid** GameObject into this Prefab.
6. Delete **Cube-boid** from the **Scene** panel.
7. With **Main Camera** selected in the **Hierarchy** panel, drag **prefab_Boid** from the **Project** panel over the public variable of `Drone` Prefab.

www.it-ebooks.info

8. Create a new cube named `wall-left`, with the following properties:
    - **Position**: (-15, 0.5, 0)
    - **Scale**: (1, 1, 20)
9. Duplicate the `wall-left` object naming the new object `wall-right`, and change the position of `wall-right` to (15, 0.5, 0).
10. Create a new cube named `wall-top`, with the following properties:
    - **Position**: (0, 0.5, 10)
    - **Scale**: (31, 1, 1)
11. Duplicate the `wall-top` object naming the new object `wall-bottom`, and change the position of `wall-bottom` to (0, 0.5, -10).
12. Finally, make the player's red cube larger, set its **Scale** to (3, 3, 3).

How it works...

The `Swarm` class contains the following three variables:

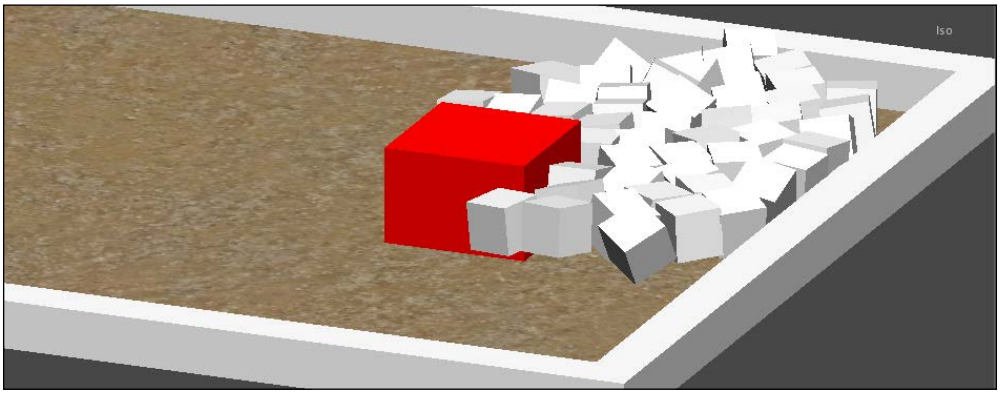
- Integer `droneCount`: This is the number of swam members to be created
- GameObject `dronePrefab`: This is a reference to the Prefab to be cloned to create swarm members
- List of `Drone` object references `drones`: This is a list of all the scripted `Drone` components inside all of the swarm GameObjects that have been created

Upon creation, as the scene starts, the `Awake()` methods of the `Swarm` class loops to create `droneCount` swarm members by repeatedly calling the `AddDrone()` method. This method instantiates a new GameObject from the Prefab and then sets variable `newDrone` to be a reference to the `Drone`-scripted object inside the new swarm member. Parameters for changing `speed`, `maxSpeed`, and `maxDirection` are set in the `newDrone` object.

Each frame `FixedUpdate()` method (used since physics is involved, so `Update()` would not be appropriate), loops through the list of `Drone` objects, calling their `UpdateVelocity()` methods, passing in the swarm center location and the average of all the swarm member velocities.

www.it-ebooks.info

The rest of this `Swarm` class is made up of two methods: one (`SwarmCenterAverage`) returns a `Vector3` object representing the average position of all the `Drone` objects, and the other (`SwarmMovementAverage`) returns a `Vector3` object representing the average velocity (movement force) of all the `Drone` objects.



The hard work is undertaken by the `Drone` class. The `SetParameters()` method stores the movement settings for the swarm into local variables for each Drone object's calculations.

The `UpdateVelocity()` method inputs `Vector3` arguments: `swarmCenterAverage` and `swarmMovementAverage`. This method then calculates the desired change in velocity for this particular `Drone` object (`swarmVelocityAdjustment`). The current movement velocity of the swarm member (`rigidbody.velocity`) has added to it `swarmVelocityAdjustment` multiplied by the `_speed` variable. The velocity of the `Drone` object is then limited to a maximum magnitude of `_maxSpeed` to prevent situations where the addition of the new velocity to the existing movement results in movement that is faster than their maximum speed.

In order to calculate `swarmVelocityAdjustment`, each `Drone` object needs to use the following to decide what to do:

- `swarmMovementAverage`
  - What is the general direction the swarm is moving?
  - This is known as **alignment**—a swarm member attempting to move in the same direction as the swarm average
- `swarmCenterAverage`
  - What is the center position of the swarm?
  - This is know as **cohesion**—a swarm member attempting to move towards the center of the swarm

www.it-ebooks.info

Variable `moveTowardsSwarmCenter` calculates a vector pointing towards the center of the swarm, by calling the `VectorTowards()` method and passing the swarm center position. The total swarm adjustment vector (`swarmVelocityAdjustment`) is the sum of this vector towards the swarm center, added to two times the average velocity of the swarm (`swarmMovementAverage`). This weighting of twice the average velocity compared to just one-times the swarm center position direction means that each `Drone` object will try to keep moving with the swarm, rather than just cluster at some center point. Without this weighting, the swarm may simply stop moving after it forms a tight grouping. The size of this weighted velocity adjustment is limited to the value of the `_maxDirectionChange` variable, which prevents `Drone` objects from changing their direction too abruptly.

The `VectorTowards()` method creates a unit vector in the direction of the given target position and then multiplies this by `_maxSpeed` to create a velocity vector of length `_maxSpeed` in the direction of the given target position. It then returns this vector with the Drone object's current velocity subtracted from it, that is, the force to be applied to make the already moving `Drone` object move towards the target position at speed equaling to `_maxSpeed`.

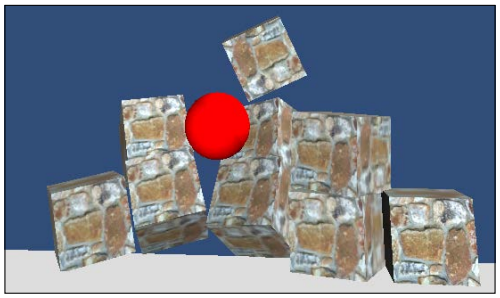
See also

- The *Controlling cube movement through player controls* recipe
- The *Controlling object look-at behavior* recipe
- The *Controlling object-to-object movements (seek, flee, follow at a distance)* recipe

Firing objects by instantiation with forward velocity

A very common game behavior for the player (or a computer-controlled non-player-character (NPC)) is to fire a projectile in the direction they are facing. This recipe presents a way to achieve this behavior.

Although there are several steps in this recipe, there are just three script classes, and most of the work is in setting up the cube wall and red sphere projectile, as seen in the following screenshot:



www.it-ebooks.info

Getting ready

In the `0423_09_05` folder, you'll find a stone image texture file called `stones.png` shown as follows:

How to do it...

To fire a projectile, perform the following steps:

1. Create a new scene and add a directional light.
2. Create a terrain sized 10 x 10, positioned at (0, 0, 0).
3. Create a new cube named `BrickCube`, sized (1, 1, 1). Apply the image texture `stones.png`, give it the tag `brick`, and add a **Rigidbody** component from **Physics** to this GameObject.
4. Create a new Prefab named `Prefab-brick`, and drag **BrickCube** into it.
5. Add the following C# script class to `Prefab-brick`:

```
// file: DestroyWhenFall.cs
using UnityEngine;
using System.Collections;

public class DestroyWhenFall : MonoBehaviour
{
  private const float MIN_Y = -1;

  void Update () {
    float y = transform.position.y;
    if( y < MIN_Y )
      Destroy( gameObject);
  }
}
```


www.it-ebooks.info

6. Create a "pyramid wall" of bricks, by placing nine instances of `BrickPrefab` in the scene at the following positions:
   - **bottom row**: (2, 0.5, 9) (3, 0.5, 9) (4, 0.5, 9) (5, 0.5, 9) (6, 0.5, 9)
   - **middle row**: (3, 1.75, 9) (4, 1.75, 9) (5, 1.75, 9)
   - **top row**: (4, 3, 9)
7. Create a new sphere named `SphereRed`, sized (1, 1, 1) with a red material, and add a **Rigidbody** component from **Physics** to this GameObject.
8. Create a new Prefab named `Prefab-sphere`, and drag **SphereRed** into the Prefab.
9. Delete **SphereRed** from the scene.
10. Add the `DestroyWhenFall` script class to `Prefab-sphere`.
11. Remove the **Main Camera** GameObject (since there is already a **Main Camera** inside the controller that you'll import in the next step).
12. Import the **Unity Character Controllers** package, and add an instance of the **First Person Controller** at (5, 1, 1).
13. Add the following C# script class to the **Main Camera** inside your **First Person Controller** (ensure **Main Camera** is selected in the **Hierarchy** panel):

```
// file: GameOverManager.cs
using UnityEngine;
using System.Collections;

public class GameOverManager : MonoBehaviour {
  private bool gameWon = false;

  void Update() {
    GameObject[] wallObjects = GameObject.FindGameObjectsWithTag("
brick");
    int numWallObjects = wallObjects.Length;

    if( numWallObjects < 1 )
      gameWon = true;
  }

  void OnGUI() {
    if( gameWon )
      GUILayout.Label("Well Done - you have destroyed the whole
wall!");
  }
}
```


www.it-ebooks.info

14. Also add the following script class to the **Main Camera** inside your **First Person Controller** (ensure **Main Camera** is selected in the **Hierarchy** panel):

```
// file: FireProjectile.cs
using UnityEngine;
using System.Collections;

public class FireProjectile : MonoBehaviour {
  public Rigidbody projectilePrefab;
  private const float MIN_Y = -1;
  private float projectileSpeed = 15f;

  /** shortest time between firing */
  public const float FIRE_DELAY = 0.25f;
  private float nextFireTime = 0f;

  void Update() {
    if( Time.time > nextFireTime )
      CheckFireKey();
  }

  void CheckFireKey() {
    if( Input.GetButton("Fire1")) {
      CreateProjectile();

      nextFireTime = Time.time + FIRE_DELAY;
    }
  }

  void CreateProjectile() {
      Rigidbody projectile = (Rigidbody)
Instantiate(projectilePrefab, transform.position, transform.
rotation);

      // create and apply velocity
    Vector3 projectileVelocity = (projectileSpeed * Vector3.
forward);
      projectile.velocity = transform.TransformDirection(
projectileVelocity );
  }
}
```

15. Ensuring **Main Camera** inside your **First Person Controller** is selected, in the **Inspector** for the **FireProjectile** scripted component, drag **Prefab-sphere** over public variable **Prefab-projectile**.

www.it-ebooks.info

How it works...

Both Prefabs (`Prefab-sphere` and `Prefab-brick`) contain rigid body components; this allows the physics engine to control instances of such objects. We can apply forces to objects with rigid bodies (for example, we can "throw" this object in a given direction), and the object will collide with and bounce off other objects. Both Prefabs also have an instance of the script class `DestroyWhenFall`—this adds the simple behavior that when either the red sphere or brick objects fall below the level of the terrain (Y=0), the objects will be destroyed. This prevents the scene from becoming filled up with all of the old sphere projectiles that have been fired. Also, it allows us to detect when the game is completed, that is, when all of the brick objects have been pushed off the terrain. Testing this condition is the responsibility of the `GameOverManager` scripted component of the **Main Camera** from **First Person Controller**. For each frame it counts the number of objects tagged with `brick`, and once that number is zero, the game completed message is displayed.

At the heart of this mini-game is the **Main Camera** scripted component `FireProjectile` of **First Person Controller**. Variable `nextFireTime` stores the time when the next projectile may be fired, and for each frame the current time is tested to see if that time has been reached. If it has, then the `CheckFireKey()` method is called. This variable is initialized to zero, so the user doesn't have to wait at all to fire the first time.

The `CheckFireKey()` method tests whether the user is pressing the Fire button at that instant. If the Fire button is pressed, the `CreateProjectile()` method is called, and the next time duration the player can fire is set to `FIRE_DELAY` seconds in the future.

Unity input settings

The default is the left mouse button, but this can be changed in the project input settings by navigating to **Edit** | **Project Settings** | **Input**.

The `CreateProjectile()` method creates a new instance of the `SpherePrefab` (via the public variable `Prefab-projectile`) at the same position and rotation of the **Main Camera** of **First Person Controller**. Note that as the camera scripted component `FireProjectile` is attached to this camera, any references to transform will retrieve the transform component of that camera.

A velocity vector for the force (movement) to be applied to the sphere projectile is calculated by multiplying the forward vector with the `projectileSpeed` variable—this variable can be tweaked to achieve the speed of firing desired for a game. To ensure the sphere projectile is moved in the same direction the camera is facing, the `TransformDirection()` method of the camera's transform component is used to set the projectile's velocity property. The result is that a new sphere projectile instance is created at the camera location, and the force is applied to make it be fired in the direction the user has oriented the camera of **First Person Controller**.

www.it-ebooks.info

There's more...

The following are some details you don't want to miss.

Addressing efficiency issues with FindGameObjectsWithTag()

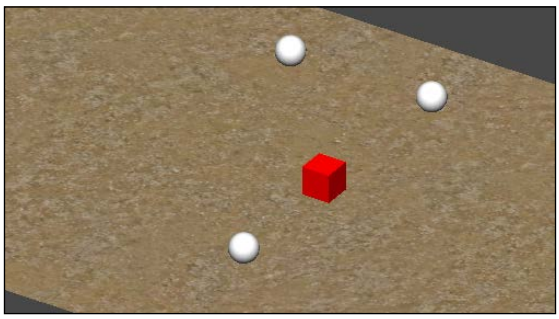
For small games, where efficiency isn't an issue, the use of `FindGameObjectsWithTag()` is fine. However, when tweaking a complex game to improve efficiency, the use of this method should be avoided as it requires a search of all GameObjects in a scene and needs to test their tag names again. A more efficient approach would be to maintain a count of the number of instances that have been created, and then to ensure that this count (list) is updated each time an object is destroyed. Thus, the count (or size of the list) is all that would be needed to be tested against for our game over condition. Another alternative to `FindGameObjectsWithTag()` is to maintain a dynamic list of all object references. See *Chapter 10*, *Improving Games with Extra Features and Optimization*, for such optimization recipes.

See also

- The *Controlling object-to-object movements (seek, flee, follow at a distance)* recipe
- The *Caching, rather than component lookups and "reflection" over objects* recipe in *Chapter 10*, *Improving Games with Extra Features and Optimization*

Finding a random spawn point

Many games make use of spawn points and waypoints. This recipe demonstrates a very common example of spawning: choosing a random spawn point and instantiating an object at that point.



Getting ready

This recipe builds upon the player-controlled cube Unity project you created in the first recipe. So make a copy of that project, open it, and then follow the steps of this recipe.

www.it-ebooks.info

How to do it...

To choose a random spawn point, perform the following steps:

1. Add the following C# script class to the **Main Camera**:

```
// file: SpawnBall
using UnityEngine;
using System.Collections;

public class SpawnBall : MonoBehaviour {
  public GameObject prefabBall;
  public GameObject[] respawns;
  public const float FIRE_DELAY = 0.25f;
  private float nextFireTime = 0f;

  private void Start(){
    respawns = GameObject.FindGameObjectsWithTag("Respawn");
  }

  void Update() {
    if( Time.time > nextFireTime )
      CheckFireKey();
  }

  void CheckFireKey() {
    if( Input.GetButton("Fire1")) {
      CreateSphere();
      nextFireTime = Time.time + FIRE_DELAY;
    }
  }

  private void CreateSphere(){
    int r = Random.Range(0, respawns.Length);
    GameObject spawnPoint = respawns[r];
    if( spawnPoint )
      Instantiate( prefab_ball, spawnPoint.transform.position,
spawnPoint.transform.rotation);
  }
}
```

2. Create a sphere sized (1, 1, 1) at position (2, 2, 2).
3. Create a new Prefab named `Prefab-ball`, and drag your sphere into it (and then delete the sphere from the **Hierarchy** panel).

www.it-ebooks.info

4. Ensuring **Main Camera** is selected, in the Inspector for the `SpawnBall` scripted component, drag **Prefab-ball** over the public variable Projectile **Prefab-ball**.
5. Create a new capsule object named `Capsule-spawnPoint` at (3, 0.5, 3), give it the tag `Respawn` (this is one of the default tags Unity provides) and uncheck its **Mesh Renderer** (so it is invisible).
6. Copy your `Capsule-spawnPoint` and move the copy to (-3, 0.5, -3).
7. Now run your game. When you click on the mouse (fire) button, a sphere should be instantiated randomly to one of the capsule locations.

How it works...

The two `Capsule-spawnPoint` objects represent candidate locations where we might wish to create an instance of our ball Prefab. When the `Start()` message is received, GameObject array `respawns` is set to the array returned from the call to `FindGameObjectsWithTag("Respawn")`. This creates an array of all objects in the scene with the tag `Respawn`, that is, the two `Capsule-spawnPoint` objects.

Variable `nextFireTime` stores the time a projectile may be fired next, and for each frame the current time is tested to see if that time has been reached. If it has, the `CheckFireKey()` method is called. This variable is initialized to zero, so the user doesn't have to wait at all to fire the first time.

The `CheckFireKey()` method tests whether at that instant the user is pressing the Fire button. If the Fire button is pressed, the `CreateSphere()` method is called. Also, the next time the player can fire is set to `FIRE_DELAY` seconds in the future.

The `CreateSphere()` method chooses a random index (`r`) in the `respawns` array and assigns the `spawnPoint` variable to the GameObject at that location in the array. It then creates a new instance of `prefab_Ball` (via the public variable) at the same position and rotation of the `spawnPoint`.

See also

- The *Finding the nearest spawn point* recipe
- The *Following waypoints in a sequence* recipe

Finding the nearest spawn point

Rather than randomly choosing a spawn point or waypoint in some cases, we will want to choose the closest point to the current position of the player or camera. This recipe builds on the previous one to move the player's red cube to the closest tagged `Respawn` object each time the Fire key is clicked.

www.it-ebooks.info

Getting ready

This recipe builds upon the previous one, so make a copy of that Unity project then follow the steps of this recipe.

How to do it...

To find the nearest spawn point, perform the following steps:

1. Remove the `SpawnBall` scripted component from the **Main Camera**.
2. Enable the **Mesh Rendered** components of your two spawn point capsules (so you can see these white capsules on the terrain).
3. Add the following C# script class to `Cube-player`:

```
// file: NearestSpawnpoint
using UnityEngine;
using System.Collections;

public class NearestSpawnpoint : MonoBehaviour {
  public GameObject[] respawns;
  public const float FIRE_DELAY = 0.25f;
  private float nextFireTime = 0f;

  private void Start(){
    respawns = GameObject.FindGameObjectsWithTag("Respawn");
  }

  void Update() {
    if( Time.time > nextFireTime )
      CheckFireKey();
  }

  void CheckFireKey() {
    if( Input.GetButton("Fire1")) {
      MoveToNewPosition();
      nextFireTime = Time.time + FIRE_DELAY;
    }
  }

  private void MoveToNewPosition(){
    transform.position = NearestSpawnpointPosition();
  }

  private Vector3 NearestSpawnpointPosition(){
```


www.it-ebooks.info

```
        if( respawns.Length < 1) return transform.position;
        Vector3 pos = respawns[0].transform.position;
        float shortestDistance = Vector3.Distance( transform.position,
pos);
        for(int i = 1; i < respawns.Length; i++){
          Vector3 newPos = respawns[i].transform.position;
          float newDist = Vector3.Distance( transform.position,
newPos);
          if( newDist < shortestDistance){
            pos = newPos;
            shortestDistance = newDist;
          }
        }

        return pos;
      }
}
```

4. Run your game. Move the red cube around with the arrow keys, and click on the mouse button. The red cube should jump to the nearest capsule spawn point.

How it works...

The `Start()` and `CheckFireKey()` methods are the same, except that `CheckFireKey()` now calls the `MoveToNewPosition()` method.

The `MoveToNewPosition()` method has a single statement, setting the position of the (player cube's) transform to the `Vector3` location returned from the `NearestSpawnpointPosition()` method.

The `NearestSpawnpointPosition()` method has the job of finding and returning the position of the spawn point closest to the current location of the (player cube's) transform. First, a check is made in case no respawn points can be found, in which case the current position of the player's cube is returned. Then, the position of first element in the `respawns` array is chosen as the first closest spawn point and is stored in the `pos` variable. The distance from the player cube to this object is stored in the `shortestDistance` variable. A `for` loop is used to loop through the remaining spawn points, and each time one is found to be closer than the distance in `shortestDistance`, then this closer position is stored into `pos` and the new closer distance stored in `shortestDistance`. Once all spawn points have been tested, the method returns the `Vector3` value in `pos`.

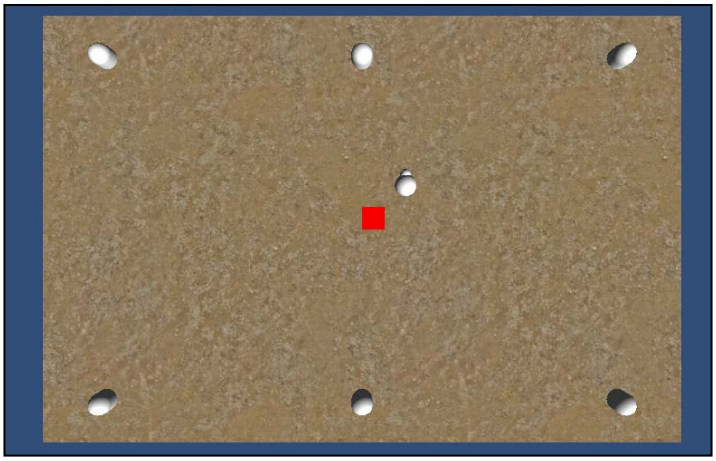
See also

- The *Finding a random spawn point* recipe
- The *Following waypoints in a sequence* recipe

www.it-ebooks.info

Following waypoints in a sequence

Waypoints are often used as a guide to make an autonomously moving NPC follow a path in a general way (but be able to respond with other directional behaviors, such as flee or seek, if predator/prey is sensed nearby). The waypoints are arranged in a sequence, so that when the character reaches or gets close to a waypoint, it will then select the next waypoint in the sequence as the target location to move towards. This recipe demonstrates an `Arrow` object moving towards a waypoint, and then when it gets close enough, choosing the next waypoint in the sequence as the new target destination. When the last waypoint has been reached, it starts again heading towards the first waypoint.



Getting ready

This recipe builds upon the player-controlled cube Unity project you created in a previous recipe. So make a copy of that project, open it, and then follow the steps of this recipe.

How to do it...

To follow waypoints in a sequence, perform the following steps:

1. Create a sphere named `Arrow`, positioned at (2, 0.5, 2) and with scale (1, 1, 1).
2. Create a second sphere, named `Sphere-small`, positioned at (2, 0.5, 2.5) and with scale (0.5, 0.5, 0.5).
3. Make `Sphere-small` a child-object of your GameObject `Arrow`.
4. Create a new capsule object named `Capsule-waypoint-0` at (-12, 0, 8), give it the tag `waypoint`.

www.it-ebooks.info

5. Copy `Capsule-spawnPoint-0` naming the copy `Capsule-spawnPoint-1` and positioning this copy at (0, 0, 8).
6. Make four more copies (named `Capsule-spawnPoint-2.5`) positioning them as follows:
   - `Capsule-spawnPoint-2`: **Position** set as (12, 0, 8)
   - `Capsule-spawnPoint-3`: **Position** set as (12, 0, -8)
   - `Capsule-spawnPoint-4`: **Position** set as (0, 0, -8)
   - `Capsule-spawnPoint-5`: **Position** set as (-12, 0, -8)
7. Add the following C# script class to GameObject `Arrow`:

```
// file: WaypointManager.cs
using UnityEngine;
using System.Collections;

public class WaypointManager : MonoBehaviour {
  public GameObject[] waypoints;

  public GameObject NextWaypoint(GameObject current){
    if( waypoints.Length < 1)
      print ("ERROR - no waypoints have been added to array!");

    // default is first in the array
    int nextIndex = 0;

    // find array index of given waypoint
    int currentIndex = -1;
    for(int i = 0; i < waypoints.Length; i++){
      if( current == waypoints[i] )
        currentIndex = i;
    }

    int lastIndex = (waypoints.Length - 1);
    if(currentIndex > -1 && currentIndex < lastIndex)
      nextIndex = currentIndex + 1;

    return waypoints[nextIndex];
  }
}
```

8. Add the following script class to GameObject `Arrow`:

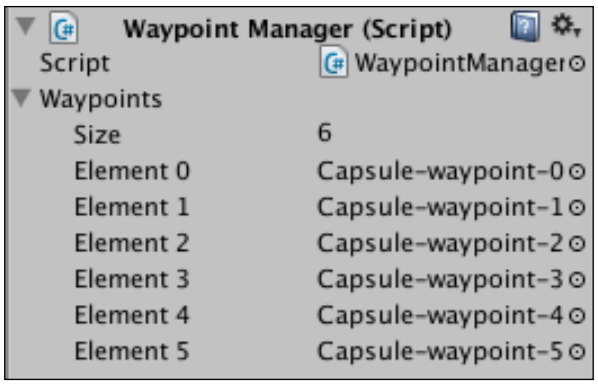
```
// file: MoveTowardsWaypoint.cs
using UnityEngine;
```


www.it-ebooks.info

```
using System.Collections;

public class MoveTowardsWaypoint : MonoBehaviour {
  public const float ARRIVE_DISTANCE = 3f;
  public float speed = 5.0F;
  private GameObject targetGO;
  private WaypointManager waypointManager;

  private void Awake(){
    waypointManager = GetComponent<WaypointManager>();
    targetGO = waypointManager.NextWaypoint(null);
  }

  private void Update () {
    transform.LookAt( targetGO.transform );
    float distance = speed * Time.deltaTime;
    Vector3 source = transform.position;
    Vector3 target = targetGO.transform.position;
    transform.position = Vector3.MoveTowards(source, target,
distance);

    if( Vector3.Distance( source, target) < ARRIVE_DISTANCE)
      targetGO = waypointManager.NextWaypoint( targetGO );
  }
}
```

9. With **Arrow** selected in the **Hierarchy** panel, in the **Waypoint Manager** scripted component in the **Inspector**, do the following:

   - Set the size of the **Waypoints** public array to **6**
   - Drag the six capsule waypoint objects into the array in sequence



www.it-ebooks.info

How it works...

The `WaypointManager` class has an array `waypoints`, and a method `NextWaypoint()`. The array is public, and via the inspector we have assigned the waypoint GameObjects in sequence to this array. The `NextWaypoint()` method takes as input a GameObject (or null), and returns a GameObject. The input is a waypoint, and the return value is the next waypoint GameObject in the array sequence, or the first in the array if the input object cannot be found.

The `MoveTowardsWaypoint` class has the following four variables:

- `ARRIVE_DISTANCE`: This is the distance from a waypoint, at which point the `Arrow` will then choose the next waypoint as its target
- `speed`: This is the distance to be travelled in each second (so as usual it will be multiplied by `Time.deltaTime` for the distance per frame)
- `targetGO`: This is the GameObject of the current target waypoint object to move towards
- `waypointManager`: This is a reference to the `WaypointManager` object in the parent (`Arrow`) GameObject

When an `Awake()` message is received, the `waypointManager` object is assigned to the `WaypointManager` in the `Arrow` object, and `targetGO` is set to be the first waypoint in the sequence (by sending the argument of `null`, the first waypoint in the array will always be returned).

The `Update()` method is responsible for making the `Arrow` object move towards the location of `targetGO`. It does this making use of the `MoveTowards()` method of `Vector3`. The final statement in the method then tests whether the current `targetGO` has been reached (distance is less than `ARRIVE_DISTANCE`). Once it has been reached, then `targetGO` is reassigned to the next waypoint in the sequence.

See also

- The *Finding a random spawn point* recipe
- The *Finding the nearest spawn point* recipe

Managing object behavior with states

Games as a whole, and individual objects or characters, can often be thought of (or modeled as) passing through different *states* or *modes*. Modeling states and changes of state (due to events or game conditions) is a very common way to manage the complexity of games and game components. In this recipe, we create a simple three-state game (game playing, game won, and game lost), using a single `GameManager` class.

www.it-ebooks.info

How to do it...

To use states to manage object behavior, perform the following step:

1. Add the following C# script class to the **Main Camera**:

```
// file: GameManager
using UnityEngine;
using System.Collections;

public class GameManager : MonoBehaviour {
  private float gameStartTime;
  private float gamePlayingTime;

  private enum GameStateType {
    STATE_GAME_PLAYING,
    STATE_GAME_WON,
    STATE_GAME_LOST,
  }

  private GameStateType currentState;

  private void Start () {
    NewGameState( GameStateType.STATE_GAME_PLAYING );
  }

  private void NewGameState(GameStateType newState) {
    // (1) state EXIT actions
    switch( currentState ) {
    case GameStateType.STATE_GAME_PLAYING:
      gameStartTime = Time.time;
      break;
    }

    // (2) change current state
    currentState = newState;

    // (3) state ENTER actions
  }

  private void Update () {
    switch( currentState ) {
    case GameStateType.STATE_GAME_PLAYING:
      StateGamePlayingUpdate();
      break;
```


www.it-ebooks.info

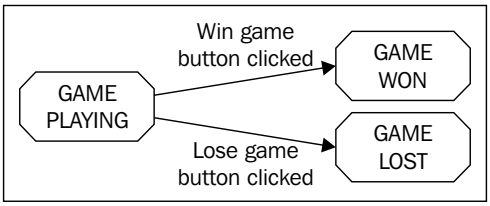
```
        case GameStateType.STATE_GAME_WON:
          StateGameWonUpdate();
          break;
        case GameStateType.STATE_GAME_LOST:
          StateGameLostUpdate();
          break;
        }
      }

      private void OnGUI () {
        switch( currentState ) {
        case GameStateType.STATE_GAME_PLAYING:
          StateGamePlayingGUI();
          break;
        case GameStateType.STATE_GAME_WON:
          StateGameWonGUI();
          break;
        case GameStateType.STATE_GAME_LOST:
          StateGameLostGUI();
          break;
        }
      }

      private void StateGamePlayingGUI() {
        GUILayout.Label("state: GAME PLAYING - time since game started
= "  + gamePlayingTime);
        bool winGameButtonClicked = GUILayout.Button("WIN the game");
        bool loseGameButtonClicked = GUILayout.Button("LOSE the
game");

        if( winGameButtonClicked )
          NewGameState( GameStateType.STATE_GAME_WON );

        if( loseGameButtonClicked )
          NewGameState( GameStateType.STATE_GAME_LOST );
      }

      private void StateGameWonGUI() {
        GUILayout.Label("state: GAME WON - game duration = " +
gamePlayingTime);
      }
```


www.it-ebooks.info

```
  private void StateGameLostGUI() {
    GUILayout.Label("state: GAME LOST - game duration = " +
gamePlayingTime);
  }

  private void StateGameWonUpdate() {   print("update - state:
GAME WON"); }
  private void StateGameLostUpdate() {  print("update - state:
GAME LOST"); }

  private void StateGamePlayingUpdate() {
    gamePlayingTime = (Time.time - gameStartTime);
    print("update - state: GAME PLAYING :: time since game started
= " + gamePlayingTime);
  }

}
```

How it works...

As can be seen in the following state chart figure, this recipe models a simple game which starts in the **GAME PLAYING** state, depending on the button clicked by the user, the game moves either into the **GAME WON** state or the **GAME LOST** state. The three possible states of the system are defined using the enumerated type `GameStateType`. The current state of the system at any point in time is stored in `currentState` variable.



Key aspects of state driven games include the following:

- When a state changes, there may be actions as a particular state is exited, then the system changes it data to represent the new state, then there may be actions as the new state is entered

  In our example, this is the responsibility of the `NewGameState()` method. The code for this method is in three parts, implementing the logic for the following steps: old state exit, set new state, and new state entered.

www.it-ebooks.info

- When the `GameManager` object receives messages (for example, every frame for `Update()` and `OnGUI()`), its behavior must be appropriate for the current state. So we see in these methods `switch` statements that call state-specific methods. For example, if the current state is `STATE_GAME_PLAYING`, then when an `Update()` message is received, the `StateGamePlayingGUI()` method will be called, and when an `OnGUI()` message is received, the `StateGamePlayingGUI()` method will be called.

However, the size and number of methods in our `GameManager` class will grow significantly with more states, and more complex game logic is needed for non-trivial games. The next recipe takes a more sophisticated approach to state-driven games, where each state has its own class.

See also

- The *Managing complex object behavior with the state pattern* recipe

Managing complex object behavior with the state pattern

The previous pattern not only illustrated the usefulness of modeling game states, but also how a game manager class can grow in size and become unmanageable. To manage the complexity of many states and complex behaviors of states, the **state pattern** has been proposed in the software development community. **Design patterns** are general purpose software component architectures that have been tried and tested and found to be good solutions to commonly occurring software system features. The key features of the state pattern are that each state is modeled by its own class, and all states inherit (are sub-classed) from a single parent state class. The states need to know about each other in order to tell the game manager to change the current state. This is a small price to pay for the division of the complexity of the overall game behaviors into separate state classes.

How to do it...

To manage an object's behavior using the state pattern architecture, perform the following steps:

1. Create a new C# script class called `GameState`:

```
// file: GameState
using UnityEngine;
using System.Collections;

public abstract class GameState : MonoBehaviour {
```


www.it-ebooks.info

```
  protected GameManager gameManager;
  protected void Awake() {
    gameManager = GetComponent<GameManager>();
  }

  public abstract void OnStateEntered();
  public abstract void OnStateExit();
  public abstract void StateUpdate();
  public abstract void StateGUI();
}
```

2. Add the following script class to the **Main Camera**:

```
// file: GameManager
using UnityEngine;
using System.Collections;

public class GameManager : MonoBehaviour {

  public StateGamePlaying stateGamePlaying;
  public StateGameWon stateGameWon;
  public StateGameLost stateGameLost;

  private GameState currentState;

  private void Awake () {
    stateGamePlaying = GetComponent<StateGamePlaying>();
    stateGameWon = GetComponent<StateGameWon>();
    stateGameLost = GetComponent<StateGameLost>();
  }

  private void Start () {
    NewGameState( stateGamePlaying );
  }

  private void Update () {
    if (currentState != null)
      currentState.StateUpdate();
  }

  private void OnGUI () {
    if (currentState != null)
```


www.it-ebooks.info

```
        currentState.StateGUI();
    }

    public void NewGameState(GameState newState)
    {
      if( null != currentState)
        currentState.OnStateExit();

      currentState = newState;
      currentState.OnStateEntered();
    }
}
```

3. Also add the following script class to the **Main Camera**:

```
// file: StateGamePlaying
using UnityEngine;
using System.Collections;

public class StateGamePlaying : GameState {
  public override void OnStateEntered(){}
  public override void OnStateExit(){}

  public override void StateGUI() {
    GUILayout.Label("state: GAME PLAYING");
    bool winGameButtonClicked = GUILayout.Button("WIN the game");
    bool loseGameButtonClicked = GUILayout.Button("LOSE the
game");

    if( winGameButtonClicked )
      gameManager.NewGameState(gameManager.stateGameWon);

    if( loseGameButtonClicked )
      gameManager.NewGameState(gameManager.stateGameLost);
  }


  public override void StateUpdate() {
    print ("StateGamePlaying::StateUpdate() - would do something
here");
  }
}
```


www.it-ebooks.info

4. Then add the following script class to the **Main Camera**:

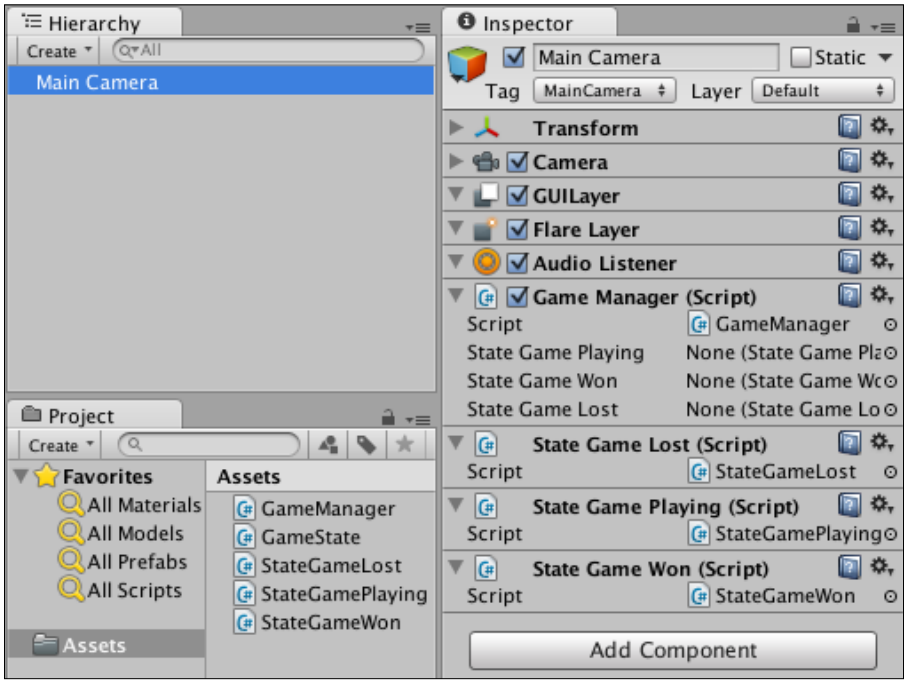
```
// file: StateGameWon
using UnityEngine;
using System.Collections;

public class StateGameWon : GameState {
  public override void OnStateEntered(){}
  public override void OnStateExit(){}

  public override void StateGUI() {
    GUILayout.Label("state: GAME WON");
  }

  public override void StateUpdate() {
    print ("StateGameWon::StateUpdate() - would do something
here");
  }
}
```

5. Add the following script class to the **Main Camera** too:

```
// file: StateGameLost
using UnityEngine;
using System.Collections;

public class StateGameLost : GameState {
  public override void OnStateEntered(){}
  public override void OnStateExit(){}

  public override void StateGUI() {
    GUILayout.Label("state: GAME LOST");
  }

  public override void StateUpdate() {
    print ("StateGameLost::StateUpdate() - would do something
here");
  }
}
```


www.it-ebooks.info

6. The following screenshot shows how the **Main Camera** has scripted components for the `GameManager` object, and each of the three state objects (`StateGameLost`, `StateGamePlaying`, and `StateGameWon`):



How it works...

The scene is very straightforward for this recipe. There is the single GameObject **Main Camera** which has four scripted object components attached to it:

- `GameManager`
- `StateGamePlaying`
- `StateGameLost`
- `StateGameWon`

www.it-ebooks.info

The `GameManager` class has three public variables, one for each of the states (`StateGamePlaying`, `StateGameLost`, and `StateGameWon`). While these public variables are unassigned before the game runs, when the `Awake()` method is executed, `GetComponent()` statements are used to assigned these variables to the three state object components in the **Main Camera**. See the information box to understand why this is necessary.

Cannot create MonoBehavior objects with the "new" keyword

If we want our state objects to be able to respond to Unity messages such as `Update()` and `OnGUI()`,the `State` class must inherit from `MonoBehavior`. However, due to the way Unity works, `MonoBehavior` objects cannot be created using the `new` keyword.

The straightforward solution to this is to attach an object of each state to the same `GameObject` as the `GameManager`. The `GameManager` object is then able to use `GetComponent()` statements to access each state of the objects.

Another alternative would be to use Prefabs and `Instantiate()`.

The `GameManager` class has one other variable `currentState`, which is a reference to the current state object at any time while the game runs (initially, it will be `null`). Since it is of the `GameState` class (the parent of all state classes), it can refer to any of the different state objects.

After `Awake()`, `GameManager` will receive a `Start()` message. This method initializes the `currentState` to be the `gamePlayingState` object.

For each frame, the `GameManager` will receive `Update()` and `OnGUI()` messages. Upon receiving these messages, `GameManager` sends `StateUpdate()` and `StateGUI()` messages to the `currentState` object. So for each frame the object for the current state of the game will execute those methods. For example, when the `currentState` is set to game playing, for each frame the `gamePlayingObject` will display the win/lose GUI buttons, and will print a console message.

When the user clicks on a button, the `gamePlayingObject` will call the `GameManager` instance's `NewState()` method, passing it the object corresponding to the new state. So if the user clicks on **Win the Game**, the `NewState()` method is passed `gameManager.stateGameWon`.

www.it-ebooks.info

A C# scripted class is defined for each state that the game needs to manage, for this example the three states `StateGamePlaying`, `StateGameWon` and `StateGameLost`. Each of these state classes is a subclass of `GameState`. `GameState` defines properties and methods that all subclass states will possess:

- The variable `gameManager`: so each state object has a link to the game manager)
- The method `Awake()`: that automatically makes variable `gameManager` refer to the `GameManager` object in the Main Camera once the scene is running)
- The four abstract methods `OnStateEntered()`, `OnStateExit()`, `StateUpdate()` and `StateGUI()`: abstract methods must have their own implementation for each subclass

See also

- The *Managing object behavior with states* recipe

www.it-ebooks.info

10

Improving Games with Extra Features and Optimization

In this chapter, we will cover the following:

- Pausing the game
- Implementing slow motion
- Implementing 3D stereography with polarized projection
- Preventing your game from running on unknown servers
- Identifying performance "bottlenecks" with code profiling
- Reducing the number of objects by destroying objects at a "death" time
- Reducing the number of enabled objects by disabling objects whenever possible
- Improving efficiency with delegates and events (and avoiding SendMessage!)
- Executing methods regularly but independent of frame rate with coroutines
- Spreading long computations over several frames with coroutines
- Caching, rather than component lookups and "reflection" over objects

Introduction

The first four recipes in this chapter provide some ideas on adding some extra features to your game (pausing, slow motion, 3D stereography, and securing online games).

The rest of the recipes in this chapter provide examples on how to investigate and improve the efficiency and performance of your game code.

www.it-ebooks.info

Pausing the game

As compelling as your next game will be, you should always let players pause it for a short break. In this recipe, we will implement a simple and effective pause screen, including controls for changing the display's quality settings.

Getting ready

For this recipe, we have prepared a package named `BallShooterGame` containing a playable scene. The package is in the `0423_10_01` folder.

How to do it...

To pause your game upon pressing the *Esc* key, perform the following steps:

1. Import the `BallShooterGame` package into your project and open the level named `RoomLevel` (from the `Scenes` folder).
2. Add the following C# Script (`PauseGame`) to **First Person Controller**:

```
// file: PauseGame.cs
using UnityEngine;
using System.Collections;

public class PauseGame : MonoBehaviour {
  public bool expensiveQualitySettings = true;
  private bool isPaused = false;
  private void Update() {
    if (Input.GetKeyDown(KeyCode.Escape)) {
      if (isPaused)
        ResumeGameMode();
      else
        PauseGameMode();
    }
  }
  private void ResumeGameMode() {
    Time.timeScale = 1.0f;
    isPaused = false;
    Screen.showCursor = false;
    GetComponent<MouseLook>().enabled = true;
  }
  private void PauseGameMode() {
    Time.timeScale = 0.0f;
    isPaused = true;
    Screen.showCursor = true;
```


www.it-ebooks.info

```
    GetComponent<MouseLook>().enabled = false;
  }
  private void OnGUI() {
    if(isPaused)
      PauseGameGUI();
  }
  private void PauseGameGUI(){
    string[] names = QualitySettings.names;
    string message = "Game Paused. Press ESC to resume or select a
new quality setting below.";
    GUILayout.BeginArea(new Rect(0, 0, Screen.width, Screen.
height));
    GUILayout.FlexibleSpace();
    GUILayout.BeginHorizontal();
    GUILayout.FlexibleSpace();
    GUILayout.BeginVertical();
    GUILayout.Label(message, GUILayout.Width(200));
    for (int i = 0; i < names.Length; i++) {
      if (GUILayout.Button(names[i],GUILayout.Width(200)))
        QualitySettings.SetQualityLevel(i,
expensiveQualitySettings);
    }
    GUILayout.EndVertical();
    GUILayout.FlexibleSpace();
    GUILayout.EndHorizontal();
    GUILayout.FlexibleSpace();
    GUILayout.EndArea();
  }
}
```

3. When you play the scene, you should be able to pause/resume the game by pressing the *Esc* key.

www.it-ebooks.info

How it works...

The value of the `Boolean` flag variable `isPaused` is used to decide what to do when *Esc* is pressed. If the game is paused, it resumes by calling `ResumeGameMode()`. If the game was not paused, it will be paused by calling the `PauseGameMode()` method.

To pause the game, we disable the `MouseLook` component, reveal the mouse cursor, and set the value of the `Time.timeScale` variable to `0`—this sets the frame rate to `0`, so if all logic is frame-based, effectively the game is paused. When the game is in pause mode (`isPaused` is `true`), `OnGUI()` will call `PauseGameGUI()` to use the opportunity to reveal the mouse cursor and display GUI buttons that activate different levels of quality settings.

To resume the game, we hide the mouse cursor, enable the `MouseLook` component, and set `Time.timeScale` back to `1.0`.

There's more...

You can always add more functionality to the Pause screen by displaying sound volume controls, save/load buttons, and so on.

Learning more about quality settings

Our code for changing quality settings is a slight modification of the example given by Unity's documentation. If you want to learn more about the subject, check it out at `docs.unity3d.com/Documentation/ScriptReference/QualitySettings.html`.

See also

- The *Implementing slow motion* recipe

Implementing slow motion

Since Remedy Entertainment's *Max Payne*, slow motion, or bullet-time, has become a popular feature in games. For example, Criterion's *Burnout* series has successfully explored the slow-motion effect in the racing genre. In this recipe, we will implement slow motion while the player presses down the right mouse button or the *Alt* key.

Getting ready

For this recipe, we will use the same prepared package as the previous recipe: `BallShooterGame` from the `0423_10_01` folder. We will also need a texture for a time progress bar, which is named `barTexture.png` and can be found in the `0423_10_02` folder.

www.it-ebooks.info

How to do it...

To implement slow-motion, perform the following steps:

1. Start a new Unity project and import the texture image file: `barTexture.png`.
2. Import the package named `BallShooterGame` into your project and open the level named `RoomLevel` (from the `Scenes` folder).
3. We need to replace all of the exiting C# `ShooterScript` with new code to implement slow motion time effects. Open `ShooterScript` and replace everything with the following code:

```
// file: ShooterScript.cs
using UnityEngine;
using System.Collections;

public class ShooterScript : MonoBehaviour
{
  public Rigidbody projectile;
  public float projectileForce = 5000.0f;
  public bool compensateBulletSpeed = true;
  public float slowMotionSpeed = 10.0f;
  public Texture2D slowMotionBar;
  public float slowMotionTotalTime = 10.0f;
  public float recoverTimeRate = 0.5f;

  private bool isSlowMotionMode = false;
  private float remainingSloMoTime;
  private float startSloMoTimestamp;
  private float elapsedTime = 0.0f;
  private float actualForce;

  private void Start() {
    Screen.showCursor = false;
    remainingSloMoTime = slowMotionTotalTime;
    actualForce = projectileForce;
  }

  private void Update() {
    CheckUserInput();

    if( isSlowMotionMode )
      SlowMotionModeActions();
    else
      NormalTimeActions();
```

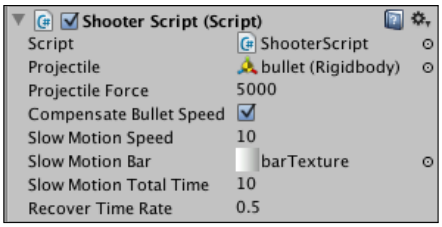

www.it-ebooks.info

```
    }

    private void CheckUserInput() {
      if (Input.GetButtonDown("Fire1")){
        Rigidbody clone = (Rigidbody) Instantiate(projectile,
Camera.main.transform.position, Camera.main.transform.rotation);
        clone.AddForce(clone.transform.forward * actualForce);
     }
      if (Input.GetButtonDown("Fire2") && remainingSloMoTime > 0)
        ActivateSloMo();

      if (Input.GetButtonUp("Fire2"))
        DeactivateSloMo();
      }

    private void SlowMotionModeActions() {
      elapsedTime = (Time.time - startSloMoTimestamp) *
slowMotionSpeed;
      remainingSloMoTime = slowMotionTotalTime - elapsedTime;

      if (elapsedTime >= slowMotionTotalTime)
        DeactivateSloMo();
    }

    private void NormalTimeActions() {
      if( remainingSloMoTime < slowMotionTotalTime )
        remainingSloMoTime += Time.deltaTime * recoverTimeRate;
      else
        remainingSloMoTime = slowMotionTotalTime;
    }

    private void ActivateSloMo() {
      Time.timeScale = 1.0f / slowMotionSpeed;
      Time.fixedDeltaTime = 0.02f / slowMotionSpeed;
      actualForce = projectileForce * slowMotionSpeed;
      isSlowMotionMode = true;

      float timeLeftToRefill = (slowMotionTotalTime -
remainingSloMoTime)  / slowMotionSpeed;
      startSloMoTimestamp = (Time.time - timeLeftToRefill);
    }

    private void DeactivateSloMo() {
      Time.timeScale = 1.0f;
```


www.it-ebooks.info

```
    Time.fixedDeltaTime = 0.02f;
    actualForce = projectileForce;
    isSlowMotionMode = false;
  }

  private void OnGUI() {
    float proportionRemaining = remainingSloMoTime /
slowMotionTotalTime;
    float barDisplayWidth = slowMotionBar.width *
proportionRemaining;

    int proportionRemaininInt = (int)(proportionRemaining * 100);
    string percentageString = proportionRemaininInt + "%";
    GUI.Label(new Rect(10, 15, 40, 20), percentageString);
    Rect newRect = new Rect(50, 10, barDisplayWidth,
slowMotionBar.height);
    GUI.DrawTexture( newRect, slowMotionBar, ScaleMode.
ScaleAndCrop, true, 10.0f);
  }
}
```

4. With the **First Person Controller** selected in the **Hierarchy** panel, drag the texture image **barTexture** into the **Inspector** for the **slowMotionBar** public variable of the **ShooterScript** component.
5. Play your game. You should be able to activate slow motion by keeping the right mouse button or the *Alt* key pressed. A progress bar will slowly shrink to indicate how much slow motion time you have remaining.



How it works...

Basically, all we need to do to have the slow motion effect is decrease the value of the `Time.timeScale` variable. In our script, we do that by dividing it by the `slowMotionSpeed` variable. We do the same to the `Time.fixedDeltaTime` variable, which updates the physics simulation of our game. The `ActivateSloMo()` method sets values for slow motion effects, and the `DeactivateSloMo()` method returns values for normal speed time.

www.it-ebooks.info

In order to make the experience more challenging, we have also implemented a sort of "energy bar" to indicate how much bullet time the player has left (the initial value is given by the `slowMotionDuration` variable). Whenever the player is not using bullet time, he has his quota filled according to the `recoverTimeRate` variable.

The GUI is implemented based on finding the proportion of slow motion time remaining, and sizing the width of the bar texture accordingly.

There's more...

The following are some suggestions for you to improve your slow motion effect even further:

Adding Motion Blur (Unity Pro only)

Motion Blur is an image effect frequently identified with slow motion. Once attached to the camera, it could be enabled and disabled through the `ActivateSloMo()` and `DeactivateSloMo()` methods respectively. More information on the Motion Blur image effect can be found at `docs.unity3d.com/Documentation/Components/script-MotionBlur.html`.

Creating sonic ambience

Max Payne famously used a strong, heavy heartbeat sound as sonic ambience. You could also try lowering the sound effects volume to convey the character "focus" when in slow motion. Plus, using audio filters on the camera could be an interesting option (provided you have Unity Pro).

See also

- The *Pausing the game* recipe

Implementing 3D stereography with polarized projection

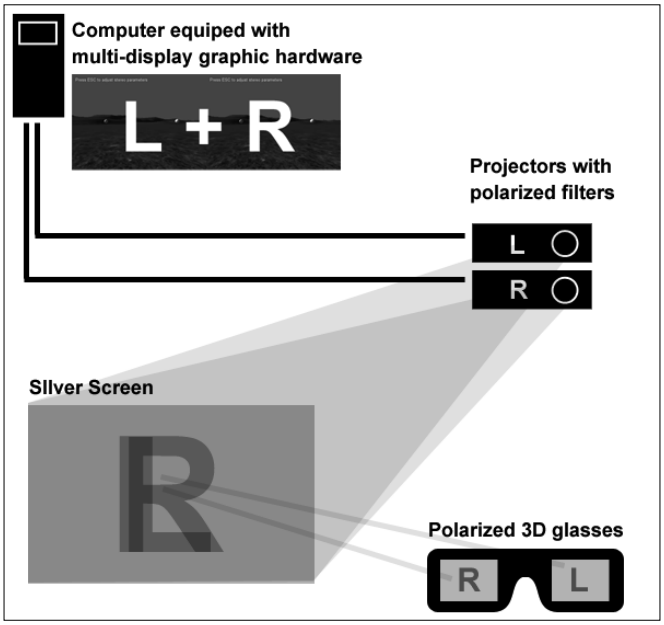
3D stereo is everywhere: from movie theaters to TV sets and portable gaming. While Unity doesn't natively support **Quad Buffer** technology in order to make active stereo implementation easy, you might achieve interesting results using passive stereoscopy through a polarized system. While this recipe might be of limited application, it might be very useful for those working with public presentations.

www.it-ebooks.info

Getting ready

For this recipe, we have prepared a package named `StereoScene` containing a playable scene including a First Person Controller, a terrain, and some basic geometry. The package is in the `0423_10_03` folder.

Although you don't need any extra hardware to try this recipe, you will of course need an adequate projection setup to test it properly. That would include a multi-display hardware such as Matrox DualHead2Go, a silver screen, two projectors with polarizing filters, and polarized 3D glasses.



How to do it...

To create side-by-side 3D stereo viewports, perform the following steps:

1. Import the `StereoScene` package into your project and open the level named `StereoCam`.
2. Add the following C# Script `StereoCam` to the **Main Camera** that is a child of the **First Person Controller** game object:

```
// file: StereoCam.cs
using UnityEngine;
using System.Collections;
```


www.it-ebooks.info

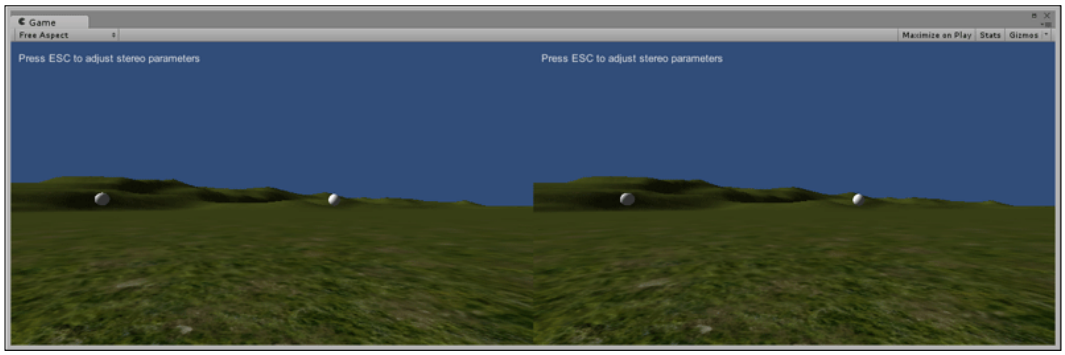
```
public class StereoCam : MonoBehaviour{
    public float parallaxDistance = 10.0f;
    public float eyeDistance = 0.05f;
    private bool showGUI = false;
    private GameObject lCamera;
    private GameObject rCamera;
    private GameObject parallax;
    void Start(){
        lCamera = new GameObject( "lCamera" );
        lCamera.AddComponent<Camera>();
        lCamera.AddComponent<GUILayer>();
        lCamera.AddComponent("FlareLayer");
        lCamera.camera.CopyFrom(Camera.main);
        rCamera = (GameObject) Instantiate(lCamera, transform.
position, transform.rotation);
        rCamera.name = "rCamera";
        lCamera.transform.parent = Camera.main.transform;
        rCamera.transform.parent = Camera.main.transform;
        parallax = new GameObject( "parallax" );
        parallax.transform.parent = Camera.main.transform;
        parallax.transform.localPosition = Vector3.zero;
        Vector3 parallaxPosition = parallax.transform.
localPosition;
        parallaxPosition.z = parallaxDistance;
        parallax.transform.localPosition = parallaxPosition;
        camera.enabled = false;
        lCamera.camera.rect = new Rect(0, 0, 0.5f, 1);
        rCamera.camera.rect = new Rect(0.5f, 0, 0.5f, 1);
        lCamera.transform.localPosition = Vector3.zero;
        rCamera.transform.localPosition = Vector3.zero;
        Vector3 cameraPosition = lCamera.transform.localPosition;
        cameraPosition.x = -eyeDistance;
   lCamera.transform.localPosition = cameraPosition;
        cameraPosition.x = eyeDistance;
        rCamera.transform.localPosition = cameraPosition;
    }
    void Update(){
        lCamera.transform.LookAt(parallax.transform);
        rCamera.transform.LookAt(parallax.transform);
        if (Input.GetKeyUp(KeyCode.Escape)){
            if (showGUI){
                showGUI = false;
            } else{
```


www.it-ebooks.info

```
                showGUI = true;
            }
        }

        if (showGUI){

            if (Input.GetKey(KeyCode.Alpha1))
                eyeDistance -= 0.001f;
            if (Input.GetKey(KeyCode.Alpha2))
                eyeDistance += 0.001f;
            if (Input.GetKey(KeyCode.Alpha3))
                parallaxDistance -= 0.01f;
            if (Input.GetKey(KeyCode.Alpha4))
                parallaxDistance += 0.01f;
            Vector3 cameraPosition = lCamera.transform.
localPosition;
            cameraPosition.x = -eyeDistance;
            lCamera.transform.localPosition = cameraPosition;
            cameraPosition.x = eyeDistance;
            rCamera.transform.localPosition = cameraPosition;
            Vector3 parallaxPosition = parallax.transform.
localPosition;
            parallaxPosition.z = parallaxDistance;
            parallax.transform.localPosition = parallaxPosition;
        }
    }
    void OnGUI(){
       for (int n = 0; n < 2; n++){
            if (showGUI){
                GUI.Label(new Rect((Screen.width * 0.5f * n) + 10,
30, 500, 20), "Eye distance: " + eyeDistance.ToString("F") + ".
Press 1 to decrease and 2 to increase.");
                GUI.Label(new Rect((Screen.width * 0.5f * n)
+ 10, 70, 500, 20), "parallax distance: " + parallaxDistance.
ToString("F") + ". Press 3 to decrease and 4 to increase.");
            }else{
                GUI.Label(new Rect((Screen.width * 0.5f * n) + 10,
10, 500, 20), "Press ESC to adjust stereo parameters");
            }
        }
    }
}
```


www.it-ebooks.info

3. Play your scene. It should now be playing in two viewports: one for left eye and the other for the right.



4. Build the executable file and run it in full-screen mode on your dual projector system, making sure it behaves as a very wide single display, not two different displays side-by-side. Left and right viewports should superimposed, and not side-by-side.

How it works...

Once attached to the **Main Camera**, our script creates three objects: two cameras (left and right) and an empty object named `parallax`. These objects are made children of the **Main Camera**, so they move and rotate accordingly. The stereographic effect is given by the distance between the left and right cameras, and also by how their focal points converge at the parallax. Also, we have used a `for` loop to make sure GUI elements are displayed on both viewports.

Since you might want to adjust the distance and parallax of your cameras, depending on your preferences and final result on your projection environment, we've made those variables adjustable (press the *Esc* key to read the instructions).

There's more...

The following are some important suggestions:

Using Render to Texture (Unity Pro only)

As an alternative to having two viewports, you could get a similar result using **Render to Texture**, as proposed by Paul Bourke at `http://paulbourke.net/stereographics/Unitystereo/`.

www.it-ebooks.info

Using anaglyph resources

In case you prefer to develop an anaglyph stereoscopic effect, there are some products you should take a look at, such as the following:

- 3D Anaglyph System, by Azer2K available at `http://u3d.as/content/azert2k/3d-anaglyph-system/1AE`
- Stereoskopix' FOV2GO at Unity's Asset Store, available at `http://u3d.as/content/stereoskopix/fov2go/2HA`

Using NVidia 3D Vision

Alternatively, you could develop your game targeting NVidia's solution 3D Vision. You can find, at the Unity Asset Store, a dedicated resource developed by Dembeta SL named Active Stereoscopic 3D for Unity at `http://u3d.as/content/dembeta-sl/active-stereoscopic-3d-for-unity/2Vt`.

Acquiring active stereo third-party software

In case you need a more robust solution for active stereo and VR, you could check out the following third-party software:

- Mechdyne sells a solution named **getReal3D** for Unity, available at `http://www.mechdyne.com`
- I'm in VR has developed **MiddleVR For Unity**, available at `http://www.imin-vr.com/middlevr-for-unity/`

Preventing your game from running on unknown servers

After all the hard work you've done to complete your web game project, it wouldn't be fair if it ended up generating traffic and income for someone else's website. In this recipe, we will create a script that prevents the main game menu from showing up unless it's hosted by an authorized server.

Getting ready

To test this recipe, you will need access to a provider where you can host the game.

www.it-ebooks.info

How to do it...

To prevent your web game from being pirated, perform the following steps:

1. Add the following C# Script to **Main Camera**:

```
// file: BlockAccess
using UnityEngine;
using System.Collections;

public class BlockAccess : MonoBehaviour {
    public bool checkDomain = true;
    public bool fullURL = true;
    public string[] DomainList;
    public string message;
    private bool illegalCopy = true;

    private void Start(){
    print (Application.absoluteURL);
        if (Application.isWebPlayer && checkDomain){
            int i = 0;
            for (i = 0; i < DomainList.Length; i++){
                if (Application.absoluteURL == DomainList[i]){
                    illegalCopy = false;
                }else if (Application.absoluteURL.
Contains(DomainList[i]) && !fullURL){
                    illegalCopy = false;
                }
            }
        }
    }

    private void OnGUI() {
        if (illegalCopy)
            GUI.Label(new Rect(Screen.width * 0.5f - 200, Screen.
height * 0.5f - 10, 400, 32), message);
        else
            Application.LoadLevel(Application.loadedLevel + 1);
    }
}
```


www.it-ebooks.info

2. Now fill out its variables. Leave the options **Check Domain** and **Full URL** checked; increase the **Size** of **Domain List** to **1** and fill out **Element 0** with the complete URL for your game. Type in the sentence `This is not a valid copy of the game` in the **Message** field, as shown in the following screenshot:

Remember to include the `.unity3d` filename and extension in the URL, and not the HTML where it is embedded.

3. Save your scene as `menu`.
4. Create a new scene and change its **Main Camera** background color to black. Save this scene as `nextLevel`.
5. Let's build the game. Navigate to **File** | **Build Settings...**, include the scenes **menu** and **nextLevel**, in that order, in the build list (scenes in **Build**). Also, select **Web Player** as your platform and click on **Build**.

How it works...

As soon as the scene starts, the script compares the actual URL of the `.unity3d` file to the ones listed in the **Block Access** component. If they don't match, the next level in the build is not loaded and a message appears on screen.

There's more...

The following is some information on how to fine-tune and customize this recipe:

Improving security by using full URLs in your Domain List

Your game will be more secure if you fill out the **Domain List** with complete URLs (such as `http://www.myDomain.com/unitygame/game.unity3d`). In fact, it's recommended that you leave the **Full URL** option selected, so your game won't be stolen and published under a URL such as `www.stolenGames.com/yourgame.html?www.myDomain.com`.

www.it-ebooks.info

Allowing redistribution with more domains

If you want your game to run from several different domains, increase **Size** and enter more URLs. Also, you can leave your game completely free of protection by leaving the **Check Domain** option unchecked.

Identifying performance "bottlenecks" with code profiling

Optimization principal 1: *Work smart not hard*. The best performance improvements are achieved when effort is focused to optimize those parts of the game that take up the most computer resources—the "bottlenecks". Code profiling involves identifying resource bottlenecks by analyzing how long different actions take to complete their tasks. For graphics-intensive operations, the **Stats** panel of the game window can be used to record frame rates and "dropped calls". For memory and processor-intensive code, Unity-Pro offers detailed, frame-by-frame, code profiling. Finally, for those who do not have Unity Pro, there is a freely available code profiling class from the Unify Community Wiki, which we will demonstrate in this recipe. Code profiling involves timing how long it takes between starting to do something, and ending it. Individual statements or loops may be profiled within a method, or the duration of a complete method may be analyzed, or the time spent between one event and some other may be checked (for example, the time between Unity to be asked to load a new scene, and that scene to then load and start running).

The Unity C# code profiler demonstrated in this recipe is by Michael Garforth, and can be found at `http://wiki.unity3d.com/index.php/Profiler`.

Getting ready

The C# script required for this recipe named `Profile.cs` can be found in the `0423_10_05` folder:

How to do it...

To profile your code performance, perform the following steps:

1. Start a new project, and import the C# script `Profile.cs`.
2. Add the following C# script class to the **Main Camera**:

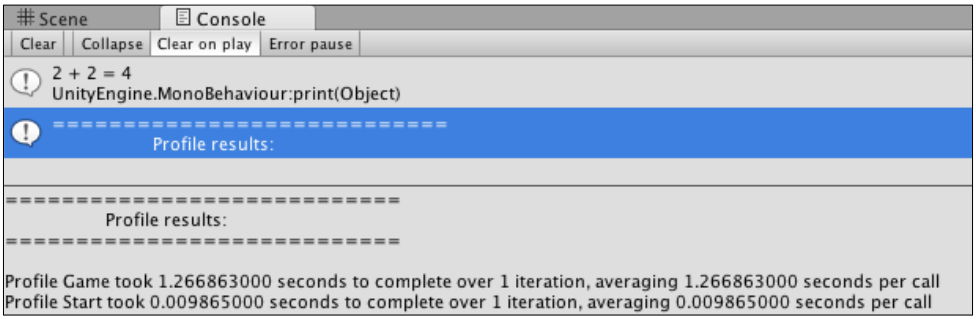
```
// file: ProfileScript.cs
using UnityEngine;

public class ProfileScript : MonoBehaviour
```


www.it-ebooks.info

```
{
  private void Awake() {
    Profile.StartProfile("Game");
  }

  private void Start(){
    Profile.StartProfile("Start");
    int answer = 2 + 2;
    print("2 + 2 = " + answer);
    Profile.EndProfile("Start");
  }

  private void OnApplicationQuit() {
    Profile.EndProfile("Game");
    Profile.PrintResults();
  }
}
```

How it works...

The `Profile` class defines three important static methods that can be called to record times for profiling: `StartProfile(<taskName>)`, `EndProfile(< taskName>)`, and `PrintResults()`. Different tasks can be timed by choosing an appropriate string (such as `Game`) and calling `StartProfile()` when they begin, and `EndProfile()` after they have completed. Finally, `PrintResults()` tells the `Profile` class to display the results of all recordings in the **Console** window:



Our `ProfileScript` object added to the **Main Camera** requests two named tasks be profiled: the simple 2 + 2 calculation in the `Start()` method (tagged **Start**); and the time between the `Awake()` method being called and the application quitting, because the user stopped game execution (tagged **Game**). As can be seen in the preceding screenshot, the profile for Start shows that the time taken was **0.009865000 seconds**; and for **Game** it was **1.266863000 seconds**.

www.it-ebooks.info

This recipe illustrates a straightforward way to measure the time taken between different points in your game. If the same string "tags" are used for `StartProfile()` and `EndProfile()` many times (for example, in a loop), then the `Profile` class will output the average time the profiled task took to complete.

There's more...

The following are some details you don't want to miss:

Unity Pro only – code profiling

If you have Unity Pro then the Unity's code **Profiler** can provide much more detailed timing analysis of your game on a frame-by-frame basis. For example (see the following screenshot), the number of objects in each frame can be analyzed (the middle line in the screenshot).

For more information, see Unity's documentation pages at `http://docs.unity3d.com/Documentation/Manual/Profiler.html`.

Reducing the number of objects by destroying objects at a "death" time

Optimization principal 2: *Minimize the number of objects in a scene*. As soon as an object is no longer needed, we should destroy it. This saves both memory and also processing resources, since Unity no longer needs to send the object `Update()` and `OnGUI()` messages, or consider collisions or physics for such objects, and so on.

However, there may be times when we wish not to destroy an object immediately, but at some known future point in time. Examples might include, after a sound has finished playing (see *Chapter 6*, *Playing and Manipulating Sounds*, for such a recipe), or perhaps the player only has a certain time to collect a bonus object before it disappears, or perhaps an object displaying a message to the player should disappear after a certain time (See *Chapter 4*, *Creating GUIs*, for such a recipe).

This recipe demonstrates how objects can be told to "start dying", and then to automatically destroy themselves after a given delay has passed.

www.it-ebooks.info

How to do it...

To destroy objects after a specified time, perform the following steps:

1. Create a cube named `Cube`.
2. Add the following script class `DeathTimeExample.cs` to `Cube`:

```
// file: DeathTimeExample.cs
using UnityEngine;
using System.Collections;

public class DeathTimeExample : MonoBehaviour {
  public float deathDelay = 4f;
  private float deathTime = -1;

  private bool menuMode = true;

  private void OnGUI(){
    if( menuMode )
      GUIMenu();
    else
      GUITimeDisplay();
  }

  private void GUITimeDisplay(){
    float timeLeft = Time.time - deathTime;
    GUILayout.Label("time left before death = " + timeLeft);

  }

  private void GUIMenu(){
    bool startDyingButtonClicked = GUILayout.Button("Start
dying");
    if( startDyingButtonClicked ){
      StartDying();
      menuMode = false;
    }
  }

  private void Update() {
    if( deathTime > 0 && Time.time > deathTime )
      Destroy( gameObject );
  }
```


www.it-ebooks.info

```
    private void StartDying() {
      deathTime = Time.time + deathDelay;
    }
  }
```

How it works...

The `OnGUI()` method initially presents a button to the user, which when clicked causes the `StartDying()` method to be called. After the button is clicked, the time left before the object destroys itself is displayed. Which of the two GUI methods is executed depends on the value of the Boolean variable `menuMode`.

The float variable `deathDelay` stores the number of seconds the object waits before destroying itself, once the decision has been made for the object to start dying. The float variable `deathTime` either has a value of `-1` (no death time set yet), or it is a positive value, which is the time we wish the object to destroy itself. When method `StartDying()` is called, it sets this `deathTime` variable to the current time plus whatever value is set in `deathDelay`.

Every frame `Update()` method calls the `CheckDeath()` method. `CheckDeath()` tests if `deathTime` greater than zero (that is, a death time has been set), and then tests whether the current time has passed the death time. If the death time has passed, then the parent `gameObject` is immediately destroyed.

When you run the scene, you'll see the Cube GameObject removed from the **Hierarchy** panel once its death time has been reached.

See also

- The *Reducing the number of enabled objects by disabling objects whenever possible* recipe

Reducing the number of enabled objects by disabling objects whenever possible

Optimization principal 3: *Minimize the number of enabled objects in a scene.* Sometimes we may not want to completely remove an object, but we can identify times when a scripted component of an object can be safely disabled. If a `Monobehaviour` script is disabled, then Unity no longer needs to send the object `Update()` and `OnGUI()` messages for each frame.

For example, if a non-player character (NPC) should only demonstrate some behavior when the player can see that character, then we only need to be executing the behavior logic when the NPC is visible—the rest of the time we can safely disable the scripted component.

www.it-ebooks.info

Unity provides the very useful events `OnBecameInvisible()` and `OnBecameVisible()`, which informs an object when it moves out of and into the visible area for one or more cameras in the scene.

This recipe illustrates the following rule of thumb: if an object has no reason to be doing actions when it cannot be seen, then we should disable that object while it cannot be seen.

How to do it...

To disable objects to reduce computer processing "workload" requirements, perform the following steps:

1. Create a new Unity project by importing the `Character Controller` package.
2. Create a new terrain.
3. Add a **3rd Person Controller** in the center of the terrain.
4. Create a new cube just in front of your **3rd Person Controller**.
5. Add the following C# script class to your cube:

```
// file: DisableWhenInvisible
using UnityEngine;
using System.Collections;

public class DisableWhenInvisible : MonoBehaviour {
  public Transform player;

    void OnBecameVisible() {
        enabled = true;
    print ("cube became visible again");
  }

    void OnBecameInvisible() {
        enabled = false;
    print ("cube became in-visible");
  }

  private void OnGUI() {
    float d = Vector3.Distance( transform.position, player.
position);
    GUILayout.Label ("distance from player to cube = " + d);
  }
}
```

6. With the cube selected in the **Hierarchy** panel, drag your **3rd Person Controller** into the **Inspector** for the public variable `player`.

www.it-ebooks.info

How it works...

When visible, the scripted `DisableWhenInvisible` component of the cube recalculates and displays the distance from itself to the transform of the **3rd Person Controller**, via the `player` variable in the `OnGUI()` method for each frame. However, when this object receives the `OnBecameInvisible()` message, the object sets its `enabled` property to `false`. This results in Unity no longer sending `Update()` and `OnGUI()` messages to the object, so the distance calculation in `OnGUI()` is no longer performed, thus reducing the game's processing workload. Upon receiving the `OnBecameVisible()` message, the `enabled` property is set back to `true`, and the object will then receive `Update()` and `OnGUI()` messages for each frame. Note that you can see the scripted component become disabled by seeing the blue "tick" in its **Inspector** checkbox disappear, if you have the cube selected in the **Hierarchy** panel, when running the game.

There's more...

The following are some details you don't want to miss:

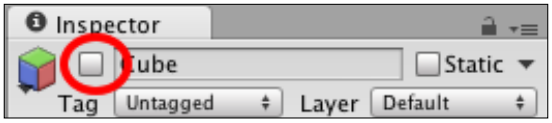
Another common case – only enable after OnTrigger()

Another common situation is that we only want a scripted component to be active if the player's character is nearby (within some minimum distance). In these situations, a sphere collider can be set up, and the scripted component can be enabled only when the player's character enters that sphere. This can be implemented using the `OnTriggerEnter()` and `OnTriggerExit()` events; for example:

```
private void OnTriggerEnter(Collider hitObjectCollider) {
if (hitObjectCollider.CompareTag("Player"))
enabled = true;
}

private void OnTriggerExit(Collider hitObjectCollider) {
if (hitObjectCollider.CompareTag("Player"))
enabled = false;
}
```

Going one step further – make the parent GameObject inactive

In some cases you can go one step further, and make the parent GameObject that contains the scripted component "inactive". This is just like deselecting the checkbox next to the GameObject in the **Inspector**.



www.it-ebooks.info

To deactivate the parent GameObject, you set its active property to `false`:

```
void OnBecameInvisible() {
  gameObject.SetActive(false);
  print ("cube became in-visible");
}
```

Note, however, that an inactive GameObject does not receive *any* messages, so it will not receive the `OnBecameVisible()` message; and this may not be appropriate for every object that is out of sight of the camera. However, when deactivating objects is appropriate, a larger performance saving is made over simply disabling a single scripted `Monobehaviour` component of a GameObject.

The only way to reactivate an inactive object is for another object to set the GameObject's active property back to `true`. The following script, when added to an active GameObject (such as terrain or controller), will display a button to demonstrate how an inactive object can be reactivated. For this script to work, the public `cubeGO` variable must be set by dragging the cube over the variable in the **Inspector**:

```
// file: MakeCubeActive.cs
using UnityEngine;
using System.Collections;

public class MakeCubeActive : MonoBehaviour {
  public GameObject cubeGO;

  private void OnGUI(){
    bool makeCubeActiveButtonClicked = GUILayout.Button("make cube
active");
    if( makeCubeActiveButtonClicked )
      cubeGO.SetActive(true);
  }
}
```

See also

- The *Reducing the number of objects by destroying objects at a "death" time* recipe

www.it-ebooks.info

Improving efficiency with delegates and events (and avoiding SendMessage!)

When events can be based on distance or collisions, we can use `OnTrigger` methods as described in the preceding recipe. When events are based on time periods, we can use coroutines, as described in a later recipe in this chapter. When there are other kinds of events, we can use C# delegates and events, as described in this recipe. These work in a similar way to `SendMessage()`, but are much more efficient, since Unity has a defined list of the objects "listening" to the broadcast events. `SendMessage()` should be avoided, since it means Unity has to analyze each scripted object ("reflect over" the object) to see whether there is a public method corresponding to the message that has been sent—this is much slower than using delegates and events.

Delegates and events implement the **publish-subscribe** (**pubsub**) design pattern. Objects can subscribe one of their methods to receive a particular type of event message from a particular publisher. In this recipe, we'll have a manager class displaying some color change buttons and publishing a new event for each button clicked; we'll also have a cube and a sphere that subscribe to the color change events, so each time a color change event is published, both the cube and the sphere should receive the event message and change their color accordingly. Publishers don't have to worry about how many objects subscribe to them at any point in time (it could be none, or 1000!), this is known as "loose coupling", since it allows different code components to be written (and maintained) independently, and this is a desirable feature of object-oriented code.

How to do it...

To implement delegates and events, perform the following steps:

1. Add the following C# script class `ColorManager` to the **Main Camera**:

```
// file: ColorManager.cs
using UnityEngine;
using System.Collections;

public class ColorManager : MonoBehaviour {
  public delegate void ColorChangeHandler(Color newColor);
  public static event ColorChangeHandler changeColorEvent;

  void OnGUI(){
    bool makeGreenButtonClicked = GUILayout.Button("make things
GREEN");
    bool makeBlueButtonClicked = GUILayout.Button("make things
BLUE");
```


www.it-ebooks.info

```
    bool makeRedButtonClicked = GUILayout.Button("make things
RED");

    if(makeGreenButtonClicked)
      PublishColorChangeEvent( Color.green );

    if(makeBlueButtonClicked)
      PublishColorChangeEvent( Color.blue );

    if(makeRedButtonClicked)
      PublishColorChangeEvent( Color.red );
  }

  private void PublishColorChangeEvent(Color newColor){
    // if there is at least one listener to this event
    if(changeColorEvent != null){
      // broadcast change color event
      changeColorEvent( newColor );
    }
  }
}
```

2. Create a cube at (0, 0, 0), and attach the following C# script class `ColorChangeListener` to it:

```
// file: ColorChangeListener.cs
using UnityEngine;
using System.Collections;

public class ColorChangeListener : MonoBehaviour {
  void OnEnable() {
    ColorManager.changeColorEvent += OnChangeColor;
  }

  private void OnDisable(){
    ColorManager.changeColorEvent -= OnChangeColor;
  }

  void OnChangeColor(Color newColor){
    renderer.sharedMaterial.color = newColor;
  }
}
```

3. Create a new material named `m_cube` and add this to your cube.

www.it-ebooks.info

4. Create a sphere at (-5, 0, 0), and attach a `ColorChangeListener` component to it.
5. Create a new material named `m_sphere` and add this to your sphere.
6. Create a directional light, so we can see the cube and sphere colors easily.

How it works...

First let's consider what we want to happen—we want the cube and sphere to both change their color when they receive an event message `OnChangeColor()` with a new color argument.

This is achieved by each object instance of the `ColorChangeListener` class (there is one present as a component of the cube and one of the sphere), subscribing their `OnChangeColor()` methods to listen for color change events published from the `ColorManager` class. Since our scripted objects may be disabled and enabled at different times, each time a scripted `ColorChangeListener` object is enabled (such as when its GameObject parent is instantiated), its `OnChangeColor()` method is added (`+=`) to the list of those subscribed to listen for color change events. Then each time `ColorChangeListener` objects are disabled, those methods are removed (`-=`) from the list of event subscribers.

When a `ColorChangeListener` object receives a color change message, its subscribed `OnChangeColor()` method is executed, and the color of the shared materials of the renderer component is changed to the received `Color` value (`green/red/blue`).

The `ColorManager` class has a public class (static) variable `changeColorEvent`, which defines an event to which Unity maintains a dynamic list of all the subscribed object methods. It is to this event that `ColorChangeListener` objects register or deregister their methods.

The `ColorManager` class displays three buttons to the user to make things green, red, and blue. When a button is clicked, the `changeColorEvent` is told to publish a new event, passing a corresponding `Color` argument to all subscribed object methods (that is, the cube and sphere).

The `ColorManager` class declares a delegate named `ColorChangeHandler`. **Delegates** define the return type (in this case `void`) and argument "signature" of methods that can be delegated (subscribed) to an event. In this case, methods must have the argument signature of a single parameter of type `Color`. Our `OnChangeColor()` method in the `ColorChangeListener` class matches this argument signature, and so it is permitted to subscribe to the `changeColorEvent` in class `ColorManager`.

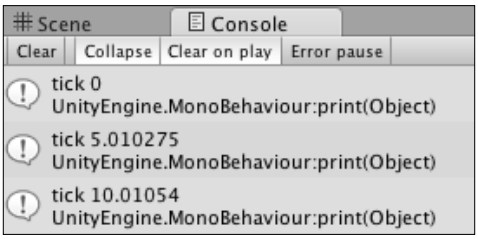
An easy-to-understand video about Unity delegates and events can be found at `http://www.youtube.com/watch?v=N2zdwKIsXJs`.

www.it-ebooks.info

Executing methods regularly but independent of frame rate with coroutines

Optimization principal 4: *Call methods as few times as possible*. While it is very simple to put logic into `Update()` and have it regularly executed each frame, we can improve the game performance by executing logic as occasionally as possible. So, if we can get away with only checking for some situations every five seconds, then great performance savings can be made to move that logic out of `Update()`.

A **coroutine** is a function that can suspend its execution until a **yield** action has completed. One kind of yield action simply waits for a given number of seconds. In this recipe, we use coroutines and `yield` to show how a method can only be executed every five seconds. This could be useful for non-player characters to decide whether they should randomly "wake up", or perhaps choose a new location to start moving towards.



How to do it...

To implement methods at regular intervals independent of the frame rate, perform the follow steps:

1. Add the following C# script class `TimedMethod` in the **Main Camera**:

```
// file: TimedMethod.cs
using UnityEngine;
using System.Collections;

public class TimedMethod : MonoBehaviour {
  private void Start() {
    StartCoroutine(Tick());
  }

  private IEnumerator Tick() {
    float delaySeconds = 5.0F;
    while (true) {
```


www.it-ebooks.info

```
            print("tick " + Time.time);
            yield return new WaitForSeconds(delaySeconds);
        }
    }
}
```

How it works...

When the `Start()` message is received, the `Tick()` method is started as a coroutine. The `Tick()` method sets the delay between executions (variable `delaySeconds`) to `5` seconds. An infinite loop is then started, where the method does its actions (in this case just printing out the time); finally, a `yield` instruction is given, which causes the method to suspend execution for the given delay of `5` seconds. After the `yield` instruction has completed, the loop will execute once again, and so on.

You may have noticed that there is no `Update()` method at all. So, although our game has logic being regularly executed, in this example, there is no logic that has to be executed every frame. Fantastic!

There's more...

The following are some details you don't want to miss:

Have different actions happening at different intervals

Coroutines can be used to have different kinds of logic being executed at different regular intervals. So logic that needs frame-by-frame execution goes into `Update()` and logic that works fine once or twice a second, might go into a coroutine with a 0.5-second delay; logic that can get away with less occasional updating can go into another coroutine with a 2- or 5-second delay, and so on. Effective and noticeable performance improvements can be found by carefully analyzing (and testing) different game logic to identify the least frequent execution that is still acceptable.

See also

- The *Spreading long computations over several frames with coroutines* recipe

www.it-ebooks.info

Spreading long computations over several frames with coroutines

Coroutines allow us to write asynchronous code—we can ask a method to go off and calculate something, but the rest of the game can keep on running without having to wait for that calculation to end. Or we can call a coroutine method for each frame from `Update()` and organize the method to complete part of a complex calculation each time it is called. When games start requiring complex computations, such as for artificial intelligence reasoning, it may not be possible to maintain acceptable game performance when trying to complete all calculations in a single frame; this is where coroutines can be an excellent solution.

This recipe illustrates how a complex calculation can be structured into several pieces, each to be completed one frame at a time.

An excellent description of coroutines (and other Unity topics) can be found on Ray Pendergraph's wikidot website at `http://raypendergraph.wikidot.com/unity-developer-s-notes#toc6`.

How to do it...

To spread computations over several frames, perform the following steps:

1. Add the following script class `SegmentedCalculation.cs` in the **Main Camera**:

```
// file: SegmentedCalculation.cs
using UnityEngine;
using System.Collections;

public class SegmentedCalculation : MonoBehaviour {
  private const int ARRAY_SIZE = 50;
  private const int SEGMENT_SIZE = 10;
  private int[] randomNumbers;

  private void Awake(){
    randomNumbers = new int[ARRAY_SIZE];
    for(int i=0; i<ARRAY_SIZE; i++){
      randomNumbers[i] = Random.Range(0, 1000);
    }

    StartCoroutine( FindMinMax() );
  }
```


www.it-ebooks.info

```
  private IEnumerator FindMinMax() {
    int min = 9999;
    int max = -1;

    for(int i=0; i<ARRAY_SIZE; i++){
      if( i % SEGMENT_SIZE == 0){
        print("frame: " + Time.frameCount + ", i:" + i + ", min:"
+ min + ", max:" + max);

        // suspend for 1 frame since we've completed another
segment
        yield return null;
      }

      if( randomNumbers[i] > max){
        max = randomNumbers[i];
      } else if( randomNumbers[i] < min){
        min = randomNumbers[i];
      }
    }

    // disable this scripted component
    print("** completed - disabling scripted component");
    enabled = false;
  }
}
```

2. When you run the scene, you should see something similar to the following:

frame: 1, i:0, min:9999, max:-1
frame: 2, i:10, min:30, max:793
frame: 3, i:20, min:30, max:892
frame: 4, i:30, min:4, max:892
frame: 5, i:40, min:4, max:905
**** completed - disabling scripted component**

How it works...

An array called `randomNumbers` of random integers is created in `Awake()`. Then the `FindMinMax()` method is started as a coroutine. The size of the array is defined by the `ARRAY_SIZE` constant, and the number of elements to process each frame by `SEGMENT_SIZE`.

www.it-ebooks.info

The `FindMinMax()` method sets initial values for `min` and `max` and begins to loop through the array. If the current index is divisible by the `SEGMENT_SIZE` (remainder 0), then we make the method display the current frame number and variable values, and suspend execution for one frame with a `yield null` statement. For each loop, the value for the current array index is compared with `min` and `max`, and those values are updated if a new minimum or maximum has been found. When the loop is completed, the scripted component disables itself.

There's more...

The following are some details you don't want to miss:

Retrieving the complete Unity log text files from your system

As well as seeing log texts in the **Console** panel, you can also access the Unity editor log text file as follows:

- Mac: `~/Library/Logs/Unity/Editor.log`
- Windows: `C:\Users\username\AppData\Local\Unity\Editor\Editor.log`

For more information about Unity logfiles, see the online manual at `http://docs.unity3d.com/Documentation/Manual/LogFiles.html`.

See also

- The *Executing methods regularly but independent of frame rate with coroutines* recipe

Caching, rather than component lookups and "reflection" over objects

Optimization principal 5: *Minimize actions requiring Unity to perform "reflection" over objects*. **Reflection** is when at run-time, Unity has to analyze objects to see whether they contain a given set of methods or components. An example of this would be the simple, useful, but slow, `FindObjectsByTag()`. Another action that slows Unity down is each time we make it look up an object's component, either explicitly using `GetComponent()`, or implicitly using a component `accessor` variable such as `transform` or `renderer`. In this recipe, we'll incrementally refactor a method, making it more efficient at each step by removing reflection and component lookup actions. The method we improve finds half the distance from the **Main Camera** (to which the script is attached) to another GameObject in the scene tagged `Player`.

www.it-ebooks.info

Getting ready

For this recipe we have provided C# script `Profile.cs` in the `0423_10_11` folder.

How to do it...

To improve code performance by caching component lookups, perform the following steps:

1. Start a new project and import the `Profile.cs` script.
2. Create a cube named `Cube-Player` at (1, 1, 1), and give it the tag `Player`.
3. Add the following C# script class `SimpleMath` to the **Main Camera**:

```
// file: SimpleMath.cs
using UnityEngine;
using System.Collections;

public class SimpleMath : MonoBehaviour {
  public float Halve(float n){
    return n / 2;
  }
}
```

4. Add the following C# script class `ProfileScript1` to the **Main Camera**:

```
// file: ProfileScript1.cs
using UnityEngine;
using System.Collections;

public class ProfileScript1 : MonoBehaviour
{
  private int ITERATIONS = 2000;

  private void Start(){
    for(int i=0; i < ITERATIONS; i++){
      FindDistanceMethod();
    }

    Profile.PrintResults();
  }

  private void FindDistanceMethod(){
    Profile.StartProfile("Method-1");
    Vector3 pos1 = transform.position;
    Transform playerTransform = GameObject.
FindGameObjectWithTag("Player").transform;
```


www.it-ebooks.info

```
    Vector3 pos2 = playerTransform.position;
    float distance = Vector3.Distance(pos1, pos2);
    SimpleMath mathObject = GetComponent<SimpleMath>();
    float halfDistance = mathObject.Halve(distance);
    Profile.EndProfile("Method-1");
  }
}
```

5. When you run the scene you should see something similar to the following:

   Profile Method-1 took 0.012487000 seconds to complete over 2000 iterations, averaging 0.000006244 seconds per call = 6.243500000 micro-seconds per call

6. The `FindGameObjectsWithTag()` method is slow, so let's fix that by adding a public `Transform` variable named `playerTransformCache`, and dragging **Cube-Player** over this variable in the **Inspector** when the **Main Camera** is selected:

   ```
   public Transform playerTransformCache;
   ```

7. Now replace your `FindDistanceMethod()` with the following, which makes use of this cached reference to the transform of `Cube-Player`, avoiding the slow object-tag reflection lookup altogether:

   ```
   private void FindDistanceMethod(){
     Profile.StartProfile("Method-2");
     Vector3 pos1 = transform.position;
     Vector3 pos2 = playerTransformCache.position;
     float distance = Vector3.Distance(pos1, pos2);
     SimpleMath mathObject = GetComponent<SimpleMath>();
     float halfDistance = mathObject.Halve(distance);
     Profile.EndProfile("Method-2");

   }
   ```

8. That should run faster (around 25-35% faster). But wait! Let's improve things some more. At the moment, to find `pos1` we are making Unity find the transform component of the **Main Camera** every time the method is called. Since the camera is not moving, let's cache this `Vector3` position as follows:

   ```
   private Vector3 pos1Cache = new Vector3();

   private void Awake(){
     pos1Cache = transform.position;
   }
   ```

9. Now replace your `FindDistanceMethod()` with the following, which makes use of this cached **Main Camera** position:

   ```
   private void FindDistanceMethod(){
     Profile.StartProfile("Method-3");
   ```


www.it-ebooks.info

```
    Vector3 pos2 = playerTransformCache.position;
    float distance = Vector3.Distance(pos1Cache, pos2);
    SimpleMath mathObject = GetComponent<SimpleMath>();
    float halfDistance = mathObject.Halve(distance);
    Profile.EndProfile("Method-3");
  }
```

10. That should improve things around 10 percent. But we can still improve things—you'll notice an explicit `GetComponent()` call to get a reference to our `mathObject`. Let's cache this scripted component reference as well, to save `GetComponent()` for each iteration. We'll declare a variable called `mathObjectCache`, and in `Awake()`, we will set it to refer to our `SimpleMath` scripted component.

```
private SimpleMath mathObjectCache;

private void Awake(){
  pos1Cache = transform.position;
  mathObjectCache = GetComponent<SimpleMath>();
}
```

11. Now replace your `FindDistanceMethod()` with the following, which makes use of this cached component reference:

```
private void FindDistanceMethod(){
  Profile.StartProfile("Method-4");
  Vector3 pos2 = playerTransformCache.position;
  float distance = Vector3.Distance(pos1Cache, pos2);
  float halfDistance = mathObjectCache.Halve(distance);
  Profile.EndProfile("Method-4");
}
```

12. After running the scene, after each improved method you should see something similar to the following, which gives us quantitative evidence of how the performance of our method is improving after each step in our refactoring process:

Profile Method-2 took 0.006818000 seconds to complete over 2000 iterations, averaging 0.000003409 seconds per call = 3.409000000 micro-seconds per call

Profile Method-3 took 0.006009000 seconds to complete over 2000 iterations, averaging 0.000003005 seconds per call = 3.004500000 micro-seconds per call

Profile Method-4 took 0.003555000 seconds to complete over 2000 iterations, averaging 0.000001778 seconds per call = 1.777500000 micro-seconds per call

www.it-ebooks.info

How it works...

This recipe illustrates how we try to cache references once, before any iteration, for variables whose value will not change. This can be done for items such as references to GameObjects and their components; and in this example, the `Vector3` position of the **Main Camera**. Through this removing of implicit and explicit component and object lookups from a method that has to be executed many times, the performance improvement is clear to see.

Two good places to learn more about Unity performance optimization techniques are from the Performance Optimization web page in the Unity Script Reference; and Unity's Joachim Ante's **Unite07**—a presentation on "Performance Optimization". Some of the recipes in this chapter had their origins from suggestions found in the following sources:

- `docs.unity3d.com/Documentation/ScriptReference/index.Performance_Optimization.html`
- `http://video.unity3d.com/video/3272748/0/unite-07-performance`

www.it-ebooks.info

www.it-ebooks.info

11
Taking Advantage of Unity Pro

In this chapter, we will cover the following:

- Dynamically focusing objects using Depth of Field
- Creating a rearview mirror
- Playing videos inside a scene
- Simulating underwater ambience with audio filters
- Loading and playing external movie files

Introduction

An awful lot can be achieved with Unity for free. However, there are some features that are only available for the "pro" version of Unity. In this chapter, we present a range of recipes illustrating some of the extra features of the "pro" version.

Dynamically focusing objects with Depth of Field

In this recipe, we will learn how to change the camera's Depth of Field focus point to the object selected by the user.

www.it-ebooks.info

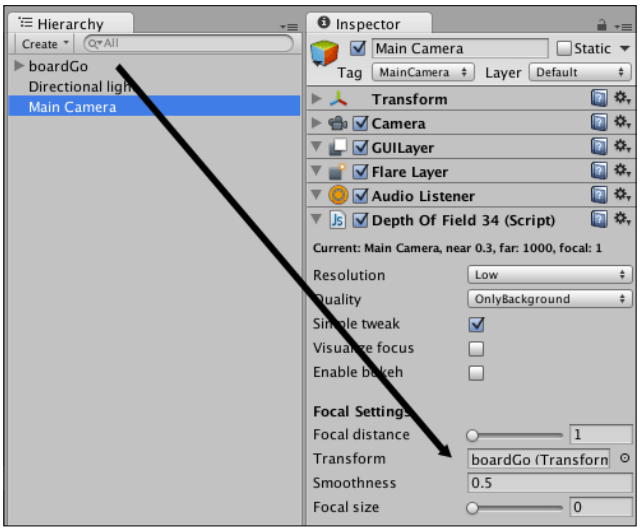
Getting ready

In order to follow this recipe, please import the `ObjectFocus` package, available in the `0423_11_01` folder, into your project. The package includes a basic scene containing a *Go* board game setup and a camera.

How to do it...

To dynamically set focus on objects using Depth of Field, perform the following steps:

1. Import the **ObjectFocus** package and open the **goBoardGame** scene, inside the **11_01** folder. Also import, if you haven't done so already, the **Package Image Effects** (**Assets** | **Import Package** | **Image Effects (Pro Only)**).
2. Select the **Main Camera** and apply the Depth of Field 3.4 image effect (**Component** | **Image Effects** | **Depth of Field (3.4)**).
3. With the **Main Camera** selected, access the **Depth of Field** component in the **Inspector** window and apply the **boardGo** game object as the **Object Focus** target. Also, change **Focal Distance** to **1**:



4. In the **Project** view, click on the **Create** drop-down menu and choose **C# Script**. Rename it to `CameraFocus` and open it in your editor.
5. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class CameraFocus : MonoBehaviour
```


www.it-ebooks.info

```
{
  private DepthOfField34 depthOfField;

  void Start(){
    depthOfField = GetComponent<DepthOfField34>();
  }
    void Update(){
        if (Input.GetButtonDown ("Fire1")) {
        Ray ray= Camera.main.ScreenPointToRay (Input.
mousePosition);
            RaycastHit hit ;
        if(depthOfField){
            if (Physics.Raycast(ray, out hit, Camera.main.
farClipPlane)) {
                depthOfField.objectFocus = hit.collider.
transform;
              }
          }
        }
    }
}
```

6. Save your script and attach it to **Main Camera**.
7. Play your scene. You should be able focus on individual tokens by clicking on them.

How it works...

The `CameraFocus` script uses a `Raycast()` function to check for colliders and, if positive, designates the collider's game object as the new focal point for the **Depth of Field 3.4** effect. Thus, it is important that these objects have colliders attached to them.

There's more...

Unity 4 includes a new, improved Depth of Field (Lens Blur, Scatter, and DX11). To use it instead of Version 3.4, please see the script named `CameraFocusScatter` in the `0423_11_01` folder.

See also

- The *Zooming a telescopic camera* recipe in *Chapter 2*, *Using Cameras*

www.it-ebooks.info

Creating a rearview mirror

In this recipe, we will take advantage of Unity Pro's ability to render camera views into textures to make a **rearview mirror**.

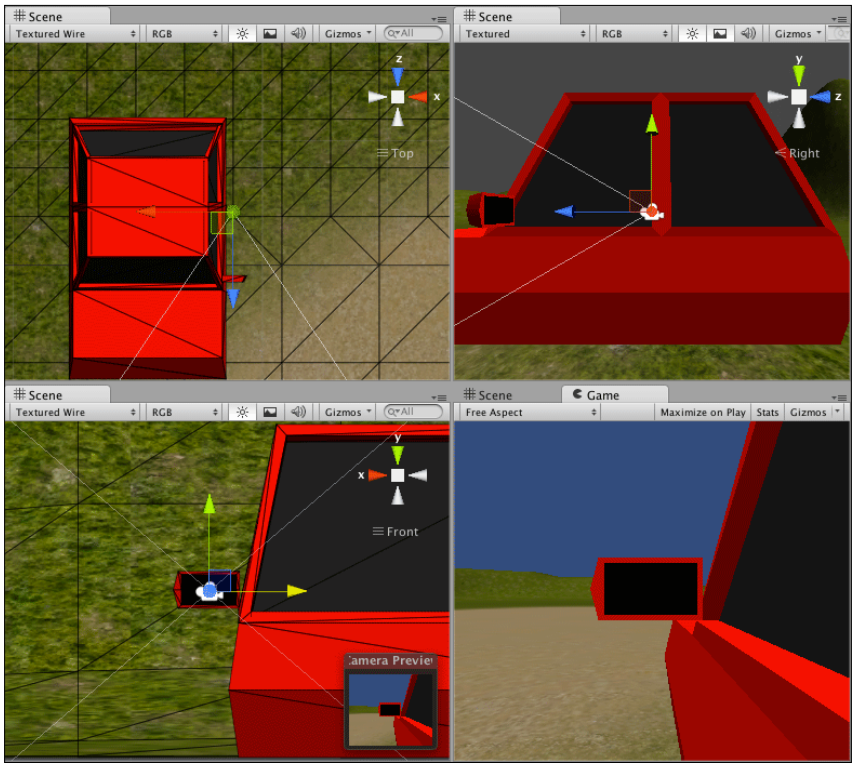
Getting ready

In order to follow this recipe, please use the example scene `mirrorPackage`, available in the Unity Package format in the `0423_11_02` folder.

How to do it...

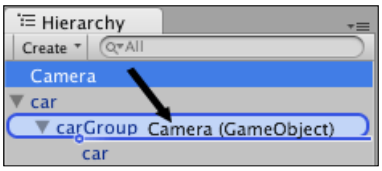
To create a rearview mirror, perform the following steps:

1. Import **mirrorPackage** and open the scene named **mirrorScene**, from the **11_02** folder.
2. Using the Transform tools (Move and Rotate), place the **Main Camera** next to the left-hand side of the car, as shown in the following screenshot. Make sure the rearview mirror is framed.

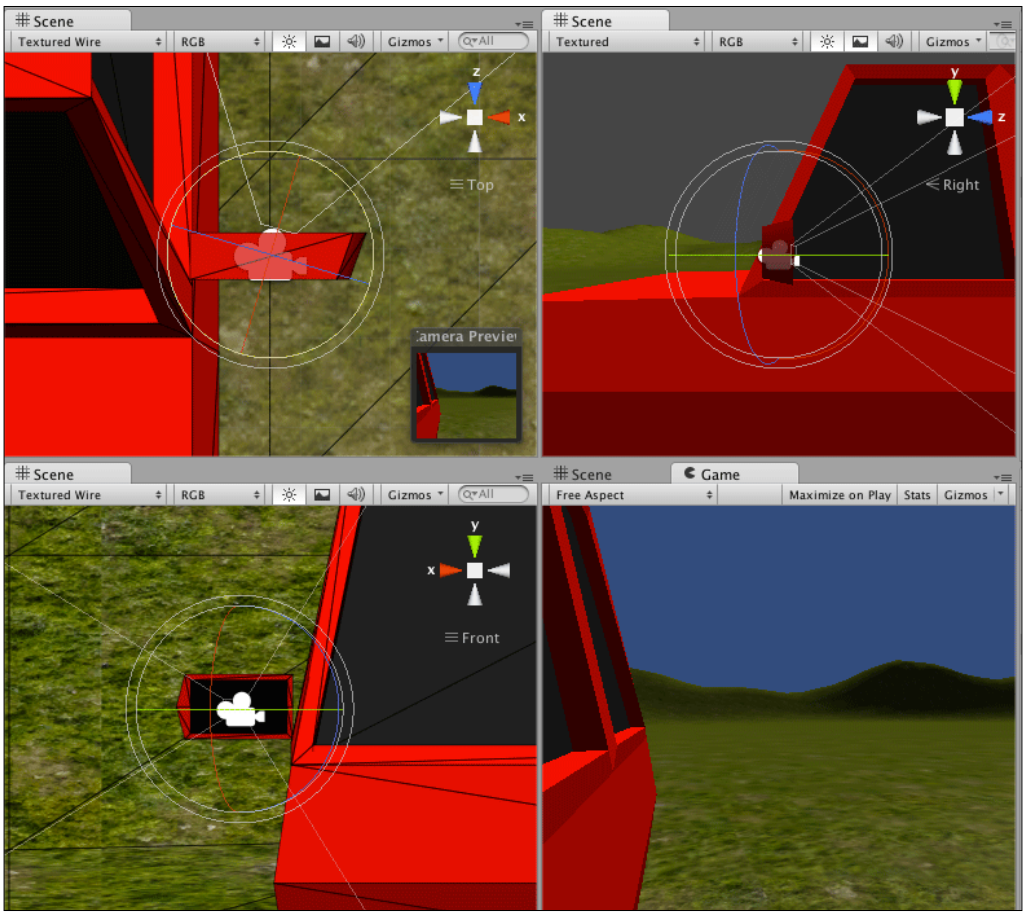


www.it-ebooks.info

3. In the **Hierarchy** view, expand the **car** object. Then, drag the camera into the **carGroup** sub-object, making it a child of the **carGroup**:

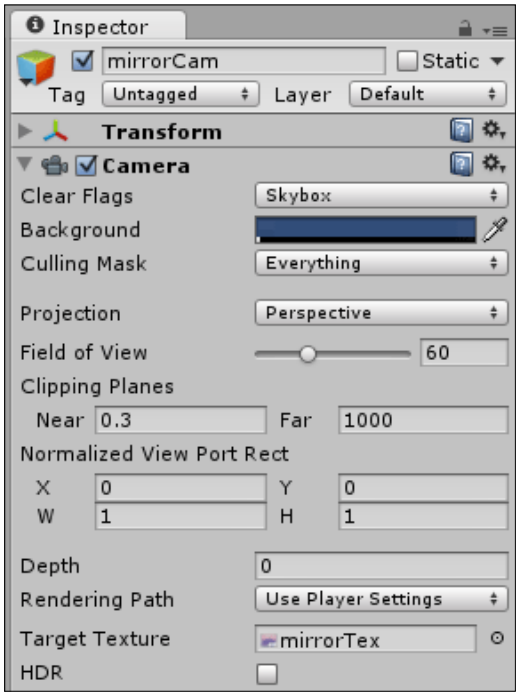


4. In the **Hierarchy** view, access the **Create** drop-down menu to add a **Camera** to the scene. Change its name to `mirrorCam` and in the **Inspector** view, disable its **Audio Listener** property.
5. Position and rotate the camera as if it is inside the rearview mirror object:



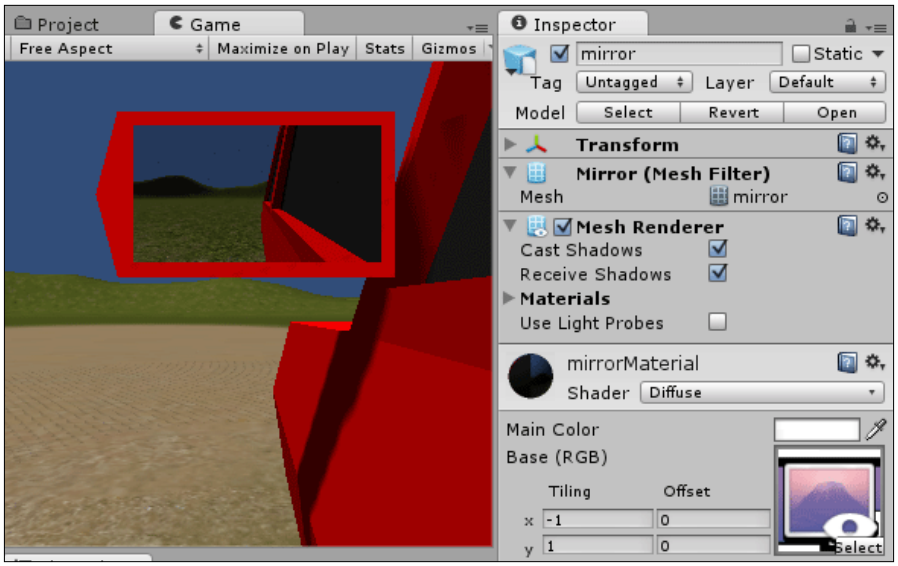
www.it-ebooks.info

6. Expand the **car** object and then the **carGroup** sub-object. Make the **mirrorCam** a child of **carGroup**.
7. Now, let's create a new material for the mirror. In the **Project** view, select the **Create** drop-down menu and choose **Material**. Name it as `mirrorMaterial`. Next, select the **Create** drop-down menu one more time and choose **Render Texture**. Rename it to `mirrorTex`.
8. In the **Inspector** view, set the **mirrorText** size to 1024 x 512.
9. Select the **mirrorCam** object. In the **Inspector** view, change the value of **Target Texture** field to **mirrorTex**.
10. Select **mirrorMaterial** and change its **Main Color** texture to **mirrorTex**. This can be done by dragging the **mirrorTex** material from the **Project** view into the **Main Color** texture slot.



11. In the **Hierarchy** window, select the **mirror** sub-object. Then apply the **mirrorMaterial** by dragging it from the **Project** view into the **Inspector** view (below its current material).
12. Now, the map must be flipped horizontally (more precisely, mirrored). In the **Project** view, select **mirrorTex** and change its **Wrap Mode** to **Repeat**.
13. Select **mirrorMaterial** and change its **x Tiling** value to `-1`:

www.it-ebooks.info



How it works...

As you have learned from the recipe, the mirror's camera view is actually being rendered into the Render Texture in real time.

There's more...

The following is a list of interesting things that can be done with Render To Texture:

Adding details

You could also make your mirror material more interesting by changing its **Shader** from **Diffuse** to **Decal**. Then you can add an RGBA texture map to it where the alpha channel would control the opaqueness of the **Decal** texture. This is a great way of adding a layer of dust or stain to the mirror.

Using Render Texture as a TV screen

You could easily adapt this recipe to make a TV or monitor screen texture. The only major change would be not changing the **x** tiling value to `-1`.

www.it-ebooks.info

Playing videos inside a scene

TV sets, projectors, monitors... If you want complex animated materials in your level, you can play video files as texture maps. In this recipe, we will learn how to apply a video texture to a cube. We will also implement a simple control scheme that plays or pauses the video when the cube is clicked.

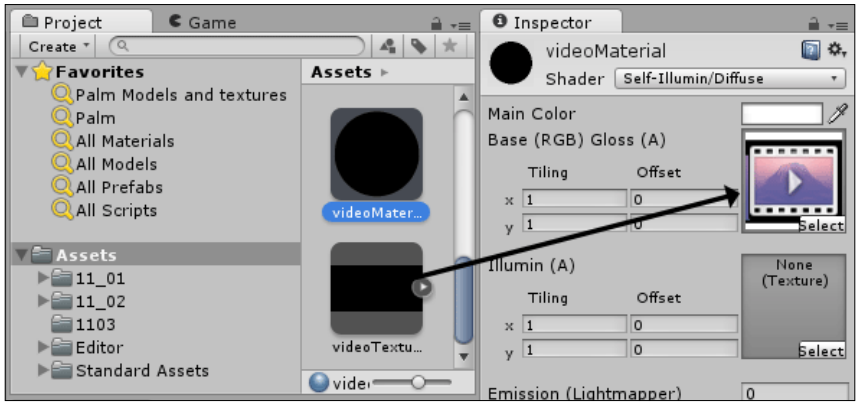
Getting ready

Unity imports video files through **Apple Quicktime**. If you don't have it installed on your machine, please download it at `http://www.apple.com/quicktime/download/`. Also, if you need a video file to follow this recipe, please use the `videoTexture.mov` file included in the `0423_11_03` folder.

How to do it...

To play a video inside a scene, perform the following steps:

1. Create a cube by navigating to **Game Object** | **Create Other** | **Cube**.
2. In the **Project** view, use the **Create** drop-down menu to create a new material. Rename it as `videoMaterial`.
3. We now need to import our video texture. Navigate to **Assets** | **Import New Asset...** and browse to your video file (it will import and convert any video file QuickTime is capable of reading).
4. Unity will convert your file to the **OGG**/**Theora** format.
5. In the **Project** view, select **videoMaterial**. Then, in the **Inspector** view, change its **Shader** to **Self-Illumin/Diffuse**.
6. Apply the **videoTexture** on the **Main Color** of **videoMaterial** by dragging it from the **Project** view into the appropriate slot:



www.it-ebooks.info

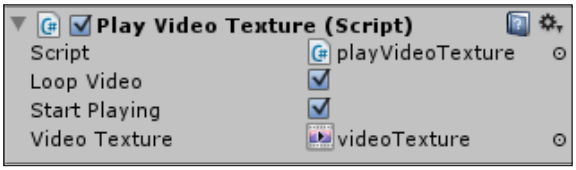
7. Apply **videoMaterial** to the cube you have created. That will give you an idea of how the video should be displayed.
8. Select the cube. Make sure there is a **Collider** component in the **Inspector**. In case there isn't one, add it by navigating to **Component** | **Physics** | **Box Collider**. Colliders are needed for mouse collision detection.
9. Now we need to create our script. In the **Project** view, click on the **Create** drop-down menu and choose **C# Script**. Rename it as `PlayVideoTexture` and open it in your editor.
10. Replace your script with the following code:

```
using UnityEngine;
using System.Collections;
[RequireComponent (typeof (AudioSource))]
public class PlayVideoTexture : MonoBehaviour {
    public bool  loopVideo = true;
    public bool  startPlaying = true;
    public MovieTexture videoTexture;
    void  Start(){
      videoTexture.loop = loopVideo;
        renderer.material.mainTexture = videoTexture;
        audio.clip = videoTexture.audioClip;
      if(startPlaying)
        ControlMovie();
    }
    void  OnMouseUp(){
       ControlMovie();
    }
    void  ControlMovie(){
      if(videoTexture.isPlaying)
        videoTexture.Pause();
      else {
        videoTexture.Play();
        audio.Play();
      }
    }
}
```

11. Save your script and apply it to the cube (in the **Hierarchy** view).

www.it-ebooks.info

12. Select the cube and, in the **Inspector** view, under the **Play Video Texture** component, select the **videoTexture** movie texture from the **Video Texture** field:



13. Test your scene. You should be able to see the movie being played in the cube face, and also pause/play it by clicking on it.

How it works...

By default, our script makes the movie texture play in the **Loop** mode. There is, however, a Boolean variable that can be changed through the **Inspector** view, where it is represented by a checkbox. Likewise, there is a checkbox that can be used to prevent the movie from playing when the level starts.

There's more...

There are some other movie texture commands and parameters that can be played with. Don't forget to check out Unity's scripting guide at `http://docs.unity3d.com/Documentation/ScriptReference/MovieTexture.html`.

Simulating underwater ambience with audio filters

It doesn't matter how great your graphics are: an underwater scene is never realistic enough without the muffled sound caused by submersion. In this recipe, we will learn how to simulate that effect whenever our player is under water.

Getting ready

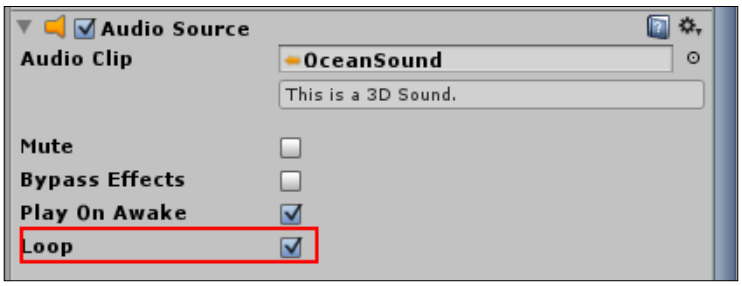
For this recipe, we have prepared a basic scene including a camera controlled by the up and down arrow keys. The scene is available inside the `AudioFilterScene` package, within the `0423_11_04` folder, where you can also find the `OceanSound.wav` audio file.

www.it-ebooks.info

How to do it...

To simulate underwater ambience, perform the following steps:

1. Import the **AudioFilterScene** package and the **OceanSound.wav** audio file into your Unity project.
2. Open the **FilterScene** level, inside the **11_04 AudioFilter** folder in the **Project** view.
3. From the **Project** view, drag the **OceanSound** asset into the **Daylight Simple Water** game object in the **Hierarchy** view.
4. In the **Hierarchy** view, select **Daylight Simple Water** and select the **Loop** option of the **Audio Source** component:



5. In the **Hierarchy** view, select the **Main Camera**. Then, add an **Audio Low Pass Filter** component to it (**Component** | **Audio** | **Audio Low Pass Filter**).
6. Change the component's **Cutoff Frequency** value to `1000`:

7. Create a new C# script and rename it as `UnderwaterSound`.
8. Open the **UnderwaterSound** script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class UnderwaterSound : MonoBehaviour
{
    public GameObject oceanObject;
    public float falloff = 1.0f;
    private AudioLowPassFilter filter;
```


www.it-ebooks.info

```
    void Start(){
      filter =  GetComponent<AudioLowPassFilter>();
      filter.enabled = false;
    }
    void Update()
    {
        if (transform.position.y < oceanObject.transform.
position.y){
            filter.enabled = true;
            filter.cutoffFrequency = 1000 - (transform.position.y
* falloff);
        }else{
            filter.enabled = false;
        }
    }
}
```

9. Save your script and attach it to the **Main Camera** by dragging it from the **Project** view into the game object in the **Hierarchy** window.
10. Drag the **Water3Example** game object from the **Project** view into the **Ocean Object** slot of the **Underwater Sound** component. Also, change the **Falloff** parameter to `20`:

11. Play the scene and press the down arrow key to move the camera down. The filter should be activated as the camera goes below the water level, dampening the sound.

How it works...

Once **Audio Low Pass Filter** has been attached, all we need to do is detect whether the camera is below the water level to enable it, also amplifying its effect by raising the **Cutoff Frequency** of the filter as the camera goes down even further.

There's more...

The following are some information on how to fine-tune and customize this recipe:

Keeping the Cutoff Frequency unchanged

To keep the **Cutoff Frequency** the same regardless of how deep in the water the camera is, just change the **Falloff** value to `0`.

www.it-ebooks.info

Adding more filters

Audio filters can be expensive in terms of processing, but that doesn't mean you can't experiment or use them in you projects. In fact, you could add a second audio filter to this recipe, such as Echo, to make the sound effect more interesting.

See also

- The *Simulating underwater ambience with Audio filters* recipe

Loading and playing external movie files

In this recipe, we will load an external image video and play it on the screen. Also, we will have the option of loading it from two different sources: the Web or **Unity Default Resources** (a library created once the game is built).

Getting ready

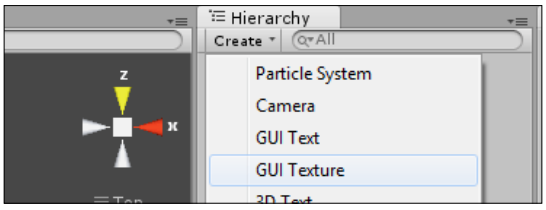
For this recipe, you'll need a video clip in OGG/Theora format. We have provided one, named `videoTexture.theora.ogg` in the `0423_11_05` folder. If you wish to use your own video files, you might need a converter. In this case, a good option is **Miro Video Converter**, available at `www.mirovideoconverter.com`.

Also, if you want to load the video clip from the Web, you will need access to a provider where you can host the files. In this case, please note that the image file should be stored in the same domain as the Unity file.

How to do it...

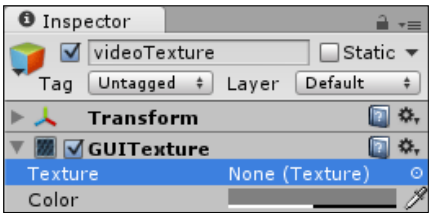
To load and play an external movie file, perform the following steps:

1. In the **Project** view, create a new folder and rename it to `Resources`.
2. Place the `videoTexture.theora.ogg` file into the `Resources` folder.
3. In the **Project** view, click on the **Create** button and add a **GUI Texture** to the scene. Then, rename it to `videoTexture`:



www.it-ebooks.info

4. Through the **Inspector** view, remove the default **GUI Texture** from **videoTexture** by selecting **None** in the **Texture** field:



5. In the **Project** view, create a new C# script and rename it to `LoadMovie`.
6. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;
using System.IO;
[RequireComponent (typeof (AudioSource))]
public class LoadMovie : MonoBehaviour  {
    public enum source {Resources_folder, Internet};
    public source VideoSource = source.Resources_folder;
    public string fileName = "";
    public string fileUrl = "";
    public bool  relativePath = false;
    private MovieTexture movieTexture;
    private string url;

    void  Start (){
            if(VideoSource != source.Resources_folder){
                GetData("external");
            } else {
                GetData("default");
        }
    }

    void  GetData ( string mode  ){
        if(mode == "web"){
            url = fileUrl;
           if(relativePath){
                url = Application.dataPath + "/" + fileUrl;
            }
            StartCoroutine(LoadWWW());
        } else if(mode == "default"){
            movieTexture = (MovieTexture) Resources.
Load(fileName);;
```

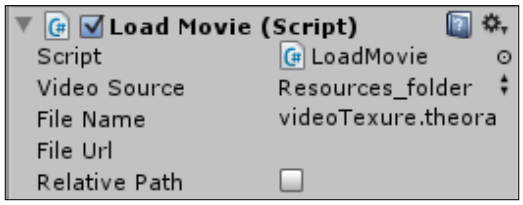

www.it-ebooks.info

```
            StartCoroutine(PlayMovie());
        }
    }

    IEnumerator  LoadWWW (){
        Debug.Log(url);
        WWW www = new WWW (url);
      yield return www;
        movieTexture = www.movie;
        StartCoroutine(PlayMovie());
    }

    IEnumerator PlayMovie(){
        while (!movieTexture.isReadyToPlay)
            yield return 0;
        Debug.Log(movieTexture);
        guiTexture.texture = movieTexture;
        audio.clip = movieTexture.audioClip;
        Debug.Log("Movie Dimensions: " + movieTexture.width +", "
+ movieTexture.height);
        float movieAspectRatio = movieTexture.width /
movieTexture.height;
        float screenAspectRatio = Screen.width / Screen.height;
        int movieWidth;
        int movieHeight;
        if (movieAspectRatio >= screenAspectRatio){
          movieWidth = Screen.width;
          movieHeight = Mathf.RoundToInt(movieWidth /
movieAspectRatio);
        } else {
          movieHeight = Screen.height;
          movieWidth = Mathf.RoundToInt(movieHeight *
movieAspectRatio);
        }
        guiTexture.pixelInset = new Rect(-(movieWidth * 0.5f),
-(movieHeight * 0.5f), movieWidth, movieHeight);
        movieTexture.Play();
        audio.Play();
    }

    void  OnGUI (){
    if(GUI.Button( new Rect(0, Screen.height - 20, 60,
20),"Play"))
      movieTexture.Play();
```


www.it-ebooks.info

```
    if(GUI.Button( new Rect(60, Screen.height - 20, 60,
20),"Pause"))
      movieTexture.Pause();
    if(GUI.Button( new Rect(120, Screen.height - 20, 60,
20),"Stop"))
      movieTexture.Stop();
    }
}
```

7. Save your script and attach it to the **videoTexture** game object by dragging it from the **Project** view to the **videoTexture** game object in the **Hierarchy** view.
8. Once the script is attached to the camera, let's check its properties. As **Video Source**, select **Resources_folder**. Fill out **File Name** by typing in the name of the video file in the `Resources` folder (in our case, `videoTexure.theora`):



9. Play the scene. The texture will be loaded and displayed on the screen.

How it works...

As soon as the scene starts, the script calls the `GetData()` function, which loads the movie file using the appropriate method, according to the preferences expressed in the component's options in the **Inspector** view.

There's more...

Next, you will find some information on how to fine-tune and customize this recipe:

Loading videos from the Internet

If you want to load the file from the Web, select **Internet** as **Video Source**, and fill out the **File Url** field with the complete file URL (unless the **Relative Path** option is checked).

www.it-ebooks.info

Index

Symbols

3D Anaglyph System
URL 321
3D stereography
active stereo third-party software, acquiring 321
anaglyph resources, using 321
implementing, with polarized projection 316-320
NVidia 3D Vision, using 321
Render to Texture, using 320
working 320
.GetAudioClip() 217
_maxDirectionChange variable 285
.text 217
.texture 217

A

accessor variable 339
ActivateRagdoll() function 182
ActivateSloMo() method 315
active stereo third-party software 321
AddDrone() method 283
AddXMLElement() method 263
adjustVelocity 278
alignment 284
anaglyph resources 321
analogue clock
displaying 98-100
screen corner clock camera, configuring 101
working 100
Android
acquiring 9
animated character
animation states, coordinating with actions 175-178
rigid props, adding to 171-174
animations
mixing, Layers and Masks used 157-163
Animation / Sound Ratio parameter
modifying 193
animation speed
audio pitch, matching to 189-193
Animator Controller
reference link 147
Apple Quicktime
URL 352
Arrive() method 278
assets
exporting 17
audio clips
preventing, from restarting 201, 202
audio filters
used, for simulating underwater ambience 354, 356
audio object
auto-destructing 202-204
audio pitch
matching, to animation speed 189-193
Audio Reverb Zone component
attaching, to audio sources 200
AudioSource component 202
audio sources
Audio Reverb Zone component, attaching to 200
Avatar 142
Avatar skeleton
configuring, on Mixamo character 142-146
Awake() method 92

www.it-ebooks.info

B

Base map 60
basicLevel package 28
Blend Trees
used, for applying Root Motion to characters 147-156
block character
completing 232
BuildXMLString() method 254
Bumped Diffuse 76

C

CalcPlayerBlipTextureRect() method 105
CameraFocus script 347
Capsule-spawnPoint objects 292
CaptureScreenshot() function 227
C# file streams
used, for loading external text files 245-247
used, for saving external text files 248, 249
changeColorEvent 334
ChangeImage() method 92
ChangeMaterial function 87
characters
animating, with Microsoft Kinect 230, 231
controlling with Microsoft Kinect, Zigfu used 227-229
CheckDeath() method 328
CheckFireKey() method 289, 292
cloudy outdoor
stimulating, cookie textures used 76-80
code performance
improving, by caching component lookups 340-343
code profiling 324
cohesion 284
ColorChangeListener component 334
ColorChangeListener objects 334
ColorManager class 332, 334
color selection dialog
creating 81, 83
working 84
colorSelector 81
compass
displaying 102-104
working 104, 105
compass class 104
complex object behavior
managing, with state pattern 302-306
component lookups
caching 339, 340
computations
spreading, over several frames 337-339
content
discovering 15
importing 16
sample project, studying 15
cookie textures
used, for stimulating cloudy outdoor 76-80
working 81
coroutine
about 335
implementing 336
countdown timer
countdown timerdisplaying, as pie-chart style clock 120, 122
countdown timerdisplaying graphically 117-119
CreateProjectile() method 289
CreateSphere() method 292
CreateXMLFromArray class 255
cross-platform problems
avoiding, with Path.Combine() method 216
cube movement
controlling, through player controls 266-268
culling, material
disabling 93
Cull Off command 94
custom mouse cursor icons
implementing 136-139
custom packages
adding, to quick list 22, 23
exporting 21, 22
importing 19
working 20
Cutoff Frequency
keeping unchanged 356
Cutout 75

D

data class
creating 254, 255
DateString() method 252

www.it-ebooks.info

DeactivateRagdoll() function 182
DeactivateSloMo() method 315
deathDelay variable 328
deathTime variable 328
delegates 334
delegates and events
implementing 332-334
Depth of Field focus point
changing, to selected object 345
design pattern 302
DestroyWhenFall script class 287
digital clock
12-hour clock, converting to 97
displaying 96, 97
digital countdown timer
displaying 116
working 117
directional logic 266
DisableWhenInvisible component 330
DrawBlip() method 109
DrawTexture() method 105
dynamic soundtrack
about 204
creating 205-210

E

EndProfile() 325
eulerAngles component 105
external audio files
loading, Unity Default Resources used 214
playing, Unity Default Resources used 214
external movie files
loading 357-360
loading, from internet 360
playing 357-360
external resource files
loading, by downloading files from internet 217, 219
loading, by manually storing files in resources folder 214-216
loading, Unity Default Resources used 212, 213
external text files
loading, by manually storing files in resources folder 216
loading, C# file streams used 245-247
loading, TextAsset public variable used 244, 245
loading, Unity Default Resources used 213
saving, C# file streams used 248, 249
external XML files
loading 249-252
parsing 249-252

F

FileReadWriteManager script 245
FindDistanceMethod() 341
FindGameObjectsWithTag()
used, for addressing efficiency issues 290
FindGameObjectsWithTag() method 341
FindMinMax() method 339
FindObjectsByTag() 339
FireProjectile 289
FixedUpdate() method 274
Flare textures 39
Flash
acquiring 9
Flash exporters 9
Flee() method 278
flocking behavior 279
forward velocity
used, for firing objects 285
fundamental properties, particle system
direction 49
energy 49
looping 49
rotation 49
speed 49

G

game
3D stereography, implementing with polarized projection 316-320
efficiency, improving 332, 333
enabled objects, reducing 328, 330
long computations, spreading with coroutines 337-339
objects, destroying at death time 326
pausing 310
performance bottlenecks, identifying with code profiling 324-326

www.it-ebooks.info

preventing, from running on unknown servers 321-323
quality settings 312
redistribution, allowing with different domains 324
security, improving 323
slow motion, implementing 312, 315
GameManager class 298, 307
game objects
creating 14
game view, Unity UI 11
GetComponent() 342
GetData() function 360
getReal3D 321
Graphical User Interface. *See* **GUIs**
Graphics.DrawTexture 57
GUI.DrawTexture 57
GUIs
about 96
creating 96
GUIs, game
analogue clock 98
compass 102
countdown timer, displaying as pie-chart style clock 120
countdown timer, displaying graphically 117
custom mouse cursor icons, implementing 136
digital clock 96
digital countdown timer 116
images, displaying for corresponding floats and ranges 112
images, displaying for corresponding integers 110
inventories, managing 130
inventory icons, for single object pickups 128
inventory texts, for single pickups 124
message fade away, creating 123
radar 106
scrollbar, controlling with mouse wheel 134

H

hierarchy view, Unity UI 11
highlightScene 88
homemade mocap
creating, by storing movements from Microsoft Kinect 232-235

I

imageArray array 92
images, for corresponding floats and ranges
displaying 112-115
images, for corresponding integers
displaying 110, 111
GUILayout, versus GUI 112
inspect camera
about 46
creating 46-48
working 48
inspector view, Unity UI 11
installation
Unity 8, 9
inventories
items, removing from List<> collection 133
managing 130-133
non-display pickups, responding to 133
inventory icons, for single object pickups
displaying 128, 129
inventory texts, for single object pickups
displaying 124-126
gameObject tag, comparing with string special method 127
IOS
acquiring 9
isPlaying property 202

L

LateUpdate() method 186
Layers and Masks
used, for mixing animations 157-163
layout, Unity UI
customizing 12
LCDMaterial 64
leaderboard
setting up, PHP/MySQL used 236-241
leaderboard data
extracting as XML 241
leaderboard scripts
securing, secret game codes used 241
LeadingZero() method 97
lens flare effect
customizing 35-39
working 39
ListToXML() method 255

www.it-ebooks.info

LoadWWW() method 217, 219
LookAt() command 186

M

maps
creating 59
materialArray 92
materials
creating 59
culling, disabling 92, 93
highlighting, at mouse over 87, 89
reflective material, creating 60
self-illuminated material, creating 64
Mecanim animation system
about 141
features 142
Root Motion, applying to characters with Blend Trees 147-156
Mecanim Example Scenes
reference link 157
member properties
projecting, with accessor methods 259, 260
menu view, Unity UI 11
message fade away
creating 123, 124
messages
facilitating, prefab used 124
methods
executing 335, 336
implementing 335
m_hours material 99
Microsoft Kinect
movements, storing from 232
used, for animating characters 230, 231
used, for controlling characters 227, 229
MiddleVR For Unity 321
mini-map
adapting, to other styles 57
creating 53-56
displaying 52
working 57
Miro Video Converter
URL 357
Mixamo 141
Mixamo character
Avatar skeleton, configuring 142-147
ragdoll physics, applying 179-182
torso, rotating to aim 183-186
m_minutes material 98
Monobehaviour script 328
Motion Blur 316
MouseLook component 312
mouseon Boolean variable 89
MoveToNewPosition() method 294
MoveTowards() method 276, 298
moveTowardsSwarmCenter variable 285
MoveTowards() Unity method
using 276
MoveTowardsWaypoint class
about 298
ARRIVE_DISTANCE variable 298
speed variable 298
targetGO variable 298
waypointManager variable 298
m_seconds material 98
multiple body parts movements
recording 236
multiple cameras
switching between 32, 34
switching, single-enabled camera used 35
switch, triggering 35

N

nearest spawn point
selecting 292-294
NearestSpawnpointPosition() method 294
NewGameState() method 301
NewState() method 307
nextImage variable 92
NextWaypoint() method 298
non-player character (NPC) object 266
normalized vector 275
NVidia 3D Vision 321

O

ObjectAtFrame class 235
object behavior
managing, with states 298-302
object group movement
controlling 279-284
object look-at behavior
controlling 270-272

www.it-ebooks.info

objects
destroying 327, 328
disabling 328-330
focusing dynamically, with Depth of Field 346, 347
firing 285-289
object-to-object movements
arrival behavior, improving 276, 277
controlling 272-275
flee target, implementing 278
OGG/Theora format 352
OnBecameInvisible() event 329
OnBecameInvisible() message 330
OnBecameVisible() event 329
OnChangeColor() method 334
OnGUI() method 97, 105, 219, 328
OnMouseDown() method 224
OnTriggerEnter() event 330
OnTriggerEnter function 174
OnTriggerExit() event 330

P

packages
about 15
importing 17, 19
parent GameObject
making inactive 330, 331
ParseScoreXML() method 252
particle effects
creating, Shuriken used 48-51
effects, adding through 52
scrub 52
working 51
Path.Combine() method 216
PauseGameGUI() 312
PauseGameMode() method 312
performance bottlenecks
identifying, with code profiling 324, 325
PHP/MySQL
used, for setting up leaderboard 236-240
picture-in-picture effect
creating 27-31
making proportional to screen size 31
position, changing 32
preventing, from updating on every frame 32
working 31
pie-chart clock style images
creating 122
PlayAnimation() method 92
PlayBackFrame() method 236
player-controlled cube
creating 266-268
player data
loading, PlayerPrefs used 223-225
loading, static properties used 219-222
saving, PlayerPrefs used 223-225
saving, static properties used 219-222
PlayerPrefs
used, loading player data 223, 225
PlayerScore object 259
PlayerXMLWriter class 263
PlaySoundIfNotPlaying() method 202
Point Light cookies
creating 81
Prefabs
about 13
creating 13
exporting, to custom package 14
working 14
preferences, Unity
editor player quality settings, changing 10
setting 9, 10
PrintResults() 325
Profile class 325
Profiler 326
ProfileScript object 325
Project browser
using 24, 25
projectileSpeed variable 289
project view, Unity UI 11
publish-subscribe (pubsub) design pattern 332

Q

Quad Buffer technology 316
Quaternion.Euler() function 100

R

radar
different blips, for different objects 110
displaying 106-108

www.it-ebooks.info

displaying, in OnGUI() method 109
working 109
radar object icons
displaying, in DrawBlip()method 109
ragdoll physics
applying, to character 179-182
working 182
random spawn point
selecting 290-292
working 292
Raycast() function 347
ReadTextFile() method 247
RecordMovements class 235
recoverTimeRate variable 316
reflection 339
Reflection Cubemap
about 60
actual scene geometry, reflecting 64
cylindrical mapping 64
shader settings 64
texture maps, using 63
working 63
reflective material
creating 60-63
Render Texture
about 57
versus, TV screen 351
Render to Texture 320
ResetGameData() method 223
resources
downloading 15
ResumeGameMode() 312
Reverb settings
creating 201
Reverb Zones
about 198
used, for simulating tunnel environment 198-200
rearview mirror
creating 348-351
details, adding 351
rigid props
adding, to animated characters 171-174
Root Motion
applying to characters, Blend Trees used 147-156
overriding, via script 164-171
rustyMetalMaterial 68

S

SaveXMLFile() method 263
scene
volume control sliders, adding to 194-197
scene view, Unity UI 11
ScoreRecordString() method 252
screenshots
saving, from game 225-227
scrollbar
controlling with mouse wheel 134, 136
Seek() method 274
SeekTarget class 276
selectableMaterial 85
SelectNodes() method 252
selectTexture 85
self-illuminated material
creating 64-67
independent maps, using 67
light, emitting over objects 67
self-illuminated shaders
highlighting with 90
SendMessage() method 332
SerializeManager() object 259
SetParameters() method 284
shaders
reference link 94
shortestDistance variable 294
Shuriken
used, for creating particle effects 48-51
single-enabled camera
using 35
slideMaterial 72
slow motion
implementing 312-315
Motion Blur, adding 316
sonic ambience, creating 316
working 315
slowMotionSpeed variable 315
sonic ambience
creating 316
sound 189
spawnPoint variable 292

www.it-ebooks.info

specular texture maps
creating 68-70
shininess, adjusting 71
specular color, changing 71
Spot Light cookies
creating 81
StartDying() method 328
Start() method 213, 219, 247
StartProfile() 325
StateGamePlayingGUI() method 302
state pattern 302
static properties
used, loading player data 219-222
Stereoskopix' FOV2GO
URL 321
StoreFrame() method 235
Swarm class 283

T

telescopic camera
zooming 43-45
TextAsset public variable
used, for loading external text files 244, 245
text-based external data 243
Texture2D image 92
textures
animating 90, 91
applying 43
combining 84-87
creating, from screen content 39, 42
using, as screenshot 43
Time.fixedDeltaTime variable 315
toolbar view, Unity UI 11
torso, Mixamo character
rotating 183-186
toss_grenade animation 179
TransformDirection() method 289
transparency texture maps
creating 72-74
transparent shaders
Bumped Diffuse, applying 76
Cutout 75
using 90
tunnel environment
simulating, with Reverb Zones 198-200

U

UI, Unity
about 10
layout 10
views 11
underwater ambience
simulating, with Audio Filters 354-356
Unite07 343
unit vector 275
Unity
about 8
characters, controlling with Microsoft Kinect 227
content, discovering 15
custom content, importing 16
downloading 8
external files, adding 14
installing 8, 9
Prefabs, creating 13
preferences, setting 9
UI, optimizing 10
Unity 4.x 8
assets, exporting to 17
Unity Asset Store
URL 20
Unity Default Resources 357
used, for loading external audio files 214
used, for loading external resource files 212, 213
used, for loading external text files 213
used, for playing external audio files 214
Unity game. *See* **game**
Unity log text files
retrieving, from system 339
Unity packages
importing 17, 18
Unity particle systems 52
Unity Pro
using 45
Unity Store
URL 9
Unity View
organizing 16
Update() method 204, 235, 274, 298

www.it-ebooks.info

UpdateVelocity() method
about 284
swarmCenterAverage argument 284
swarmMovementAverage argument 284

V

VectorTowards() method 285
videos
playing, in scene 352-354
views, Unity UI
about 11
game 11
hierarchy 11
inspector 11
menu 11
project 11
scene 11
toolbar 11
volume control sliders
adding, to scene 194-197

W

WaypointManager class 298
waypoints
following, in sequence 295-297
WelcomeGUI() method 224
WriteEndDocument() method 254
WriteEndElement() method 254
WriteStartDocument() method 254
WriteStartElement() method 254
WriteTextFile() method 249
WriteWhiteSpace() 254
WWW class 217

X

XML data files
retrieving, from web 252
XMLDocument.Save() method 261
XML strings
newlines, adding 254
XML text data
creating, through serialization 256-259
creating, XML Writer used 252-254
XML text files
creating 261-263
XMLWriter
used, for creating XML test data
manually 252-254

Y

yield 335

Z

ZigEngageSingleUser component 232
Zigfu 227
zooming effect 45

www.it-ebooks.info

www.it-ebooks.info

Thank you for buying
Unity 4.x Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

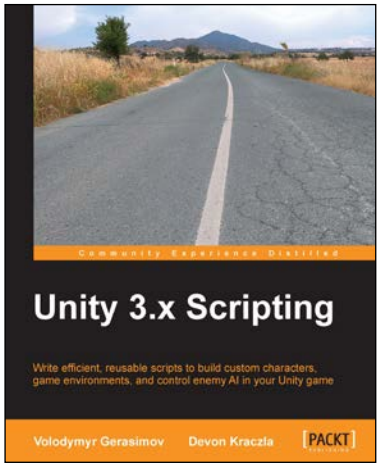
Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

www.it-ebooks.info

[PACKT]
PUBLISHING



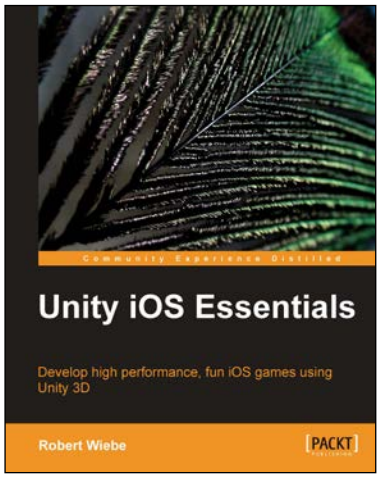
Unity 3.x Scripting

ISBN: 978-1-84969-230-4 Paperback: 292 pages

Write efficient, reusable scripts to build custom characters, game environments, and control enemy AI in your Unity game

1. Make your characters interact with buttons and program triggered action sequences
2. Create custom characters and code dynamic objects and players' interaction with them
3. Synchronize movement of character and environmental objects

---



Unity iOS Essentials

ISBN: 978-1-84969-182-6 Paperback: 358 pages

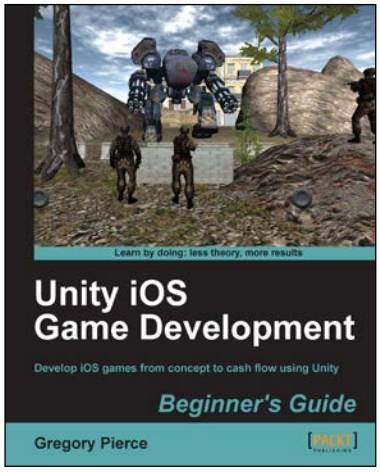
Develop high performance, fun iOS games using Unity 3D

1. Learn key strategies and follow practical guidelines for creating Unity 3D games for iOS devices.
2. Learn how to plan your game levels to optimize performance on iOS devices using advanced game concepts.
3. Full of tips, scripts, shaders, and complete Unity 3D projects to guide you through game creation on iOS from start to finish.

Please check **www.PacktPub.com** for information on our titles

www.it-ebooks.info

[PACKT] PUBLISHING



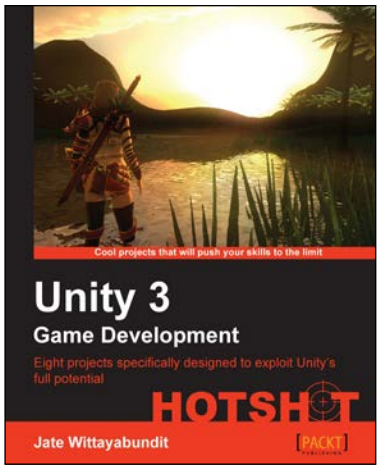
Unity iOS Game Development Beginners Guide

ISBN: 978-1-84969-040-9 Paperback: 314 pages

Develop iOS games from concept to cash flow using Unity

1. Dive straight into game development with no previous Unity or iOS experience
2. Work through the entire lifecycle of developing games for iOS
3. Add multiplayer, input controls, debugging, in app and micro payments to your game

---



Unity 3 Game Development Hotshot

ISBN: 978-1-84969-112-3 Paperback: 380 pages

Eight projects specifically designed to exploit Unity's full potential

1. Cool, fun, advanced aspects of Unity Game Development, from creating a rocket launcher to building your own destructible game world
2. Master advanced Unity techniques such as surface shader programming and AI programming
3. Full of coding samples, diagrams, tips and tricks to keep your code organized, and completed art assets with clear step-by-step examples and instructions

Please check **www.PacktPub.com** for information on our titles

www.it-ebooks.info